TECHNISCHE
UNIVERSITÄT
DARMSTADT

Incrementalization of Analyses for Next Generation IDEs

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor–Ingenieurs (Dr.-Ing.)

vorgelegt von

Diplom Informatiker Sven Kloppenburg

geboren in Darmstadt

Referent:	Prof. Dr.–Ing. Mira Mezini
Korreferent:	Prof. Dr. rer. nat. Andy Schürr
Datum der Einreichung:	14. 11. 2008
Datum der mündlichen Prüfung:	16. 1. 2009

Darmstadt 2009

D17

Abstract

To support developers in their day–to–day work, Integrated Develoment Environments (IDEs) incorporate more and more ways to help developers focus on the inherent complexities of developing increasingly larger software systems. The complexity of developing large software systems can be categorized [24] into inherent complexity that stems from the complexity of the problem domain and accidential complexity that stems from the shortcomings of the tools and methods used to tackle the problem. For example: To reduce the complexity of having to know exactly, which methods a certain class provides, IDEs offer autocompletion. To alert developers to errors and potential errors in their use of the programming language, IDEs connect the lists of warnings and errors with their source locations. To ease navigation in bigger projects, structural views of the program, such as the type hierarchy are presented. Development environments thus enable developers to be more productive and help them to find bugs earlier in the development cycle by using codified expert knowledge.

In these environments, static anlyses are used to extract information from the program under development. Static analyses detect properties of programs without running them. In the past, static analyses were mostly integrated into compilers with the goal to check for errors and to produce faster or smaller code. Integrating static analyses into the IDE opens up new areas of application. Among these are domain specific analyses, optional type systems and checks for structural properties. Domain specific analyses check properties specific to the program under development. For example, that the use of a framework conforms to the specifications. Optional type systems [22] are type systems that do not influence the runtime semantics. This allows to have multiple type systems (e.g. confined types [135] and the builtin Java type system) to coexist and to be checked by static analyses.

If these analyses are available to developers, a wider range of software defects can be detected. By integrating the analyses into the IDE, faster and better feedback can be delivered. This enables developers to incorporate the analyses in their daily workflow, as it preserves the immediacy [24].

To gain full advantage of IDE integration, the analyses need to be integrated into the incremental build process of the IDE and the rulebases should be modularly modifiable to fit the program under inspection [24]. One example for an open, modular approach to achieve this is Magellan [55]. Magellan is a build process integrated open static analysis platform that tackles the problems of integrating static analyses with the IDE and in particular with the incremental build process. To benefit from this integration, analyses running on such platforms need to work in an incremental fashion.

In this thesis, approaches for incrementalizing static analyses for integrated open static analysis platforms are analyzed. Incrementalizing a static analysis means, that the analyses uses the result from a previous build and the changes made to the program as additional input to reconcile the result from the previous build. The result is equal to an analysis of the full build.

The approaches can be categorized into manual incrementalization and automatic incrementalization. Manual incrementalization uses a general purpose language, such as Java, to implement a static analysis that achieves the incrementalization using a special purpose algorithm. Automatic incrementalization means, that the analysis is written with the full build in mind, and the underlying mechanisms of the language or framework has a builtin mechanism to reconcile the results for the changed program.

Currently, incremental analyses are developed in an ad hoc fashion, choosing the approach the developer is most familiar with. If the appraoch taken is not the best for the problem at hand, then either the development will take longer or the analysis will run slower then necessary. To investigate the properties of analyses that influence the recommended appraoches to incrementalization, three static analyses have been selected. The analyses were implemented twice; once using the manual approach and once using the automatic approach.

The three selected analysis represent analyses that check for data flow properties, control flow properties and structural properties of the inspected program.

The analysis that checks for data flow properties searches for violations of an optional type system for confined types.

The analysis that checks for control flow properties incrementally computes the call graph using the rapid type analysis (RTA).

Finally, the static analysis that checks for structural properties searches for violations of structural dependencies between concerns in the program.

The results indicate, that analyses incorporating query engines to be used by the user of the analysis need to use automatic incrementalization at least for this purpose. Analyses that can be configured only in narrow, predictable ways lend themselves to manual incrementalization. Then the domain knowledge allows for domainspecific optimizations that cannot easily be integrated into the frameworks for automatic incrementalization.

Zusammenfassung

Um Entwickler in ihrer täglichen Arbeit zu unterstützen, integrieren integrierte Entwicklungsumgebungen (IDEs) zunehmend mehr Hilfsmittel, die es Entwicklern erlauben, sich auf die inhärente Komplexität der Entwicklung zunehmend größerer Software Systeme zu konzentrieren.

Die Komplexität der Entwicklung dieser Systeme wird unterteilt [24] in inhärente Komplexität die aus der Komplexität der Problemstellung stammt, sowie accidentielle[1] Komplexität die von der Unzulänglichkeit der verwendeten Werkzeuge und Methoden kommt und daher durch bessere Werkzeuge beseitigt werden kann. So bieten IDEs automatische Vervollständigung an, damit Entwickler sich nicht die genaue Schreibweise von Methoden merken müssen. Um Entwickler auf (potentielle) Fehler im Gebrauch der Programmiersprache hinzuweisen, werden Fehlermeldungen in der IDE mit dem Quelltext verknüpft. Um die Navigation in Projekten zu erleichtern bieten IDEs strukturelle Ansichten des Programms, wie z. B. die Typhierarchie an. IDEs ermöglichen Entwicklern, produktiver zu sein und Fehler früher zu finden, in dem sie kodifiziertes Expertenwissen nutzen.

In IDEs werden statische Analysen benutzt, um Informationen aus dem Programm in Entwicklung zu extrahieren. Statische Analysen entdecken Eigenschaften von Programmen, ohne diese auszuführen. In der Vergangenheit waren statische Analysen meist in Compilern integriert, um Fehler zu finden und kleineren oder schnelleren Code zu produzieren. Werden statische Analysen in IDEs integriert, öffnen sich ihnen neue Anwendungsgebiete. Unter diesen sind domainspezifische Analysen, optionale Typsysteme und das Überprüfen struktureller Eigenschaften. Optionale Typsysteme [22] sind Typsysteme die die Laufzeitsemantik nicht verändern. Das erlaubt es, mehrere Typsysteme (zum Beispiel *confined types* [135] und das Java Typsystem) zu kombinieren und von statischen Analysen prüfen zu lassen.

Wenn diese Analysen Entwicklern zur Verfügung stehen, kann ein breiterer Bereich von Softwaredefekten erkannt werden. Durch das Integrieren der Analysen in die Entwicklungsumgebung kann dem Entwickler schneller und besser Rückmeldung gegeben werden. Das erlaubt es den Entwicklern, die Analysen in ihren alltäglichen Arbeitsablauf zu integrieren, da die Unmittelbarkeit der Rückmeldungen gewahrt bleibt [24].

Um den vollen Vorteil der IDE Integration zu erreichen, müssen die Analysen zum Einen in den inkrementellen Übersetzungsvorgang eingebettet werden und zum Anderen müssen die Analysen die ausgeführt werden an das untersuchte Programm anpassbar sein [24]. Ein Beispiel für einen offenen,

[1] Das "accidentielle" ist [58] das Unwesentliche, Wechselende, Äussere, Zufällige

modularen Ansatz um dies zu erreichen ist Magellan [55]. Magellan ist eine offene statische Analyseplattform die in den inkrementellen Übersetzungsvorgang integriert ist und die es ermöglicht, statische Analysen in IDEs und insbesondere in den inkrementellen Übersetzungsvorgang zu integrieren.

In dieser Arbeit werden Ansätze zur Inkrementalisierung statischer Analysen für integrierte, offene statische Analyseplattformen untersucht. Eine statische Analyse zu inkrementalisieren bedeutet, dass die Analyse die Ergebnisse eines vorhergenden Übersetzunsvorgangs und die Änderungen am Programm nutzt, um das Analyseergebnis an den aktuellen Zustand des Programms anzugleichen. Das Analyseergebnis ist dann äquivalent zu einer kompletten Analyse des Programms im aktuellen Zustand.

Die Ansätze hierzu können in manuelle und automatische Inkrementalisierung eingeteilt werden. Manuelle Inkrementalisierung nutzt eine universelle Programmiersprache, wie beispielsweise Java, um eine statische Analyse zu implementieren, die die Inkrementalisierung in einem spezialisierten Algorithmus verwirklicht. Bei automatischer Inkrementalisierung wird die Analyse geschrieben, wie für die komplette Analyse, da Die zugrundeliegende Sprache beziehungsweise Framework einen Mechanismus anbietet, um die Analyseergebnisse an die Programmänderungen anzupassen.

Gegenwärtig werden inkrementelle Analysen ad hoc entwickelt, mittels dem Ansatz, der dem Entwickler am vertrautesten ist. Wenn aber der Ansatz nicht der am besten geeignetste für das Problem ist, wird die Entwicklungszeit oder die Laufzeit der Analyse länger sein als notwendig. Um die Eigenschaften von Analysen zu untersuchen, die die Wahl des Ansatzes beeinflussen, wurden drei Analysen ausgewählt. Diese Analysen wurden je einmal mit dem manuellen und dem automatischen Ansatz implementiert.

Die ausgewählten Analysen repräsentieren Analysen, die den Daten- und den Kontrollfluss untersuchen, sowie Analysen die strukturelle Eigenschaften überprüfen. Die Analyse die Dateflusseigenschaften überprüft, sucht Verletzungen des optionalen Typsystems *confined types*. Die Analyse die den Kontrollfluss untersucht, erstellt und wartet einen intraprozeduralen *call graph* mit Hilfe der *rapid type analysis* (schnelle Typanalyse, RTA). Die Analyse die strukturelle Eigenschaften prüft, sucht nach Verletzungen von strukturellen Abhängigkeiten zwischen Belangen (*concerns*) im Programm.

Die Ergebnisse deuten darauf hin, dass Analysen, die Abfragemechanismen (*query engines*) beinhalten, zumindest für diesen Teil automatische Inkrementalisierung nutzen sollten. Analysen, die sich nur in einfacher, Vorhersagbarer Weise konfigurieren lassen, eignen sich eher für manuelle Inkrementalisierung. Dann kann Wissen über das Fachgebiet der Problemstellung Optimierungen ermöglichen, die sich nicht ohne weiteres in Umgebungen für automatische Inkrementalisierung integrieren lassen.

Contents

List of Figures

List of Tables

List of Listings

Preface

Completing the work on this dissertation marks a major milestone in my scientific career as well as in my life.

First and foremost my thanks go to Mira Mezini, my supervisor. She gave me the chance to pursue scientific work in earnest and saw me through, even and especially in times of doubt. Thank you.

I also want to thank my colleagues. They supported me and this work with lots of coffee and discussions, which resulted in many valuable insights. From this group I want to highlight Michael Eichberg, who collaborated with me on most papers. The following list of colleagues is sorted alphabetically, as each of them allowed me to learn from them (among the learnings are: how to be a better scientist, how to teach more effectively, how do get a lot of work done and have fun during the time, how to meet deadlines, and sometimes even what not to do): Ivica Aracic, Christoph Bockisch, Marcel Bruch, Vasian Cepa, Anis Charfi, Tom Dinkelaker, Vaidas Gasiunas, Mathias Halbach, Michael Haupt, Wolfgang Heenes, Slim Kallel, Karl Klose, Klaus Ostermann, Shadi Rifai, Thorsten Schäfer, Tobias Schuh and Andreas Sewe.

Special thanks go to Gudrun Harris as she helped me navigate the bue-rocratic and logistic challenges during my work at the university.

All errors that remain in this work are mine. I am grateful that Martin Girschick, Christoph Bockisch, Patrick Jäger and Michael Eichberg took the time to proofread parts of this work and managed to reduce the amount of errors quite a bit.

Last but not least, thanks also go to my beloved family and friends, especially my parents and to Stephanie, my wonderful wive. Due to their support, I was able to finish this big project successfully.

I want to thank all of you that made this possible.

Chapter 1

Overview

This chapter presents an overview about the topics covered in this thesis. After giving an introduction in the next section, Section 1.2 summarizes the aims of the thesis. Section 1.3 discusses the contributions and Section 1.4 details the organization of the thesis.

1.1 Introduction

Software projects are getting ever more large and complex. With the increasing size and complexity of the projects, the possibilities to introduce errors multiply. The longer errors remain undetected, the more difficult they are to remove when detected [20]. Therefore, it is important to support developers in fixing errors as early as possible, which is the time, when the code is written.

Code is usually written using an integrated development environment (IDE). An IDE is an integrated set of tools to develop software that comprises at least editor, compiler, linker and debugger, which are presented with a unified user interface. To support developers in their day–to–day work, IDEs incorporate more and more mechanisms that reduce the complexity of developing large software systems.

The complexity of developing large software systems can be categorized into *inherent* and *accidental* complexity [24]. Inherent complexity stems from the complexity of the problem domain and comprises the complexity inherent in the algorithms and data structures that are necessary to accurately represent the problem. Accidental complexity stems from the shortcomings of the tools and methods used to tackle the problem and thus can be reduced by improving the tools. Examples for IDE–features that reduce accidental

complexity are:

- Auto-completion for method and class names reduces the complexity of having to know exactly, which methods a certain class provides.
- Lists of errors and warnings alert developers to errors and potential errors in their use of the programming language.
- Structural views of the program—such as the type hierarchy—ease the navigation in bigger projects.

With features like these, IDEs help to shift the focus from the accidental to the inherent complexities of developing increasingly larger software systems. IDEs thus enable developers to be more productive and help them to find bugs earlier in the development cycle.

Most of the mechanisms used to improve IDEs make use of static analyses. Static analyses provide interesting insights into properties of programs, without running them. Traditionally, static analyses are either used to produce faster or smaller code or to check properties that are independent of an application's domain, such as array index out of bounds, null-pointer dereferences, unused code or buffer overflows. Thus, these analyses are often integrated in compilers. Recently, attention is shifting towards domain and project specific analyses e.g. for Web and EJB applications [52, 93, 113, 114], to check the correct usage of specific APIs [14], to find violations of security constraints [96], and to enforce design or programming guidelines [82]. If analyses like these are available to developers, a wider range of software defects can be detected.

By integrating analyses into the development environment, faster and better feedback can be delivered. The accidental complexity of building large software systems can be reduced by the integration of more and better static analyses into IDEs.

If developers are to incorporate the use of analyses in their daily workflow, immediate feedback from the analyses is a necessary prerequisite. With immediate feedback, the developers flow of work can continue uninterrupted as there is no need to change context or to wait for analysis results.

Modern IDEs support the development of large software systems by allowing *incremental builds*. These are build processes during which only those artifacts (i.e. source code and other files, e.g. configurations files) are rebuilt that depend on changed artifacts. As an example, consider the process of Java development using the Eclipse[1] IDE. The Java Development Tools

[1] `http://www.eclipse.org`

(JDT) are part of Eclipse and provide an incremental compiler which is integrated with the editor. The compiler errors or warnings are shown attached to the source location of the error in the editor. This provides immediate feedback for developers, which helps to keep them in the flow of the problem they try to solve. The traditional way of showing the compiler errors in separate tools after explicit build commands breaks this flow.

Incremental building has obvious performance benefits for projects with hundreds or thousands of files (called resources in Eclipse), where only a tiny fraction is changed for any given build. The technical challenge for incremental building is to determine exactly what needs to be rebuilt. To continue the example, the JDT uses a "last build state", maintained internally by the builder, to do a build based on the changes in the project since the last build. In addition to the changed resources, the builder keeps track of dependent files and recompiles them only when necessary. For example, the internal state maintained by the Java builder includes things like a dependency graph and a list of compilation problems reported. This information is used during incremental builds to identify which classes need to be recompiled in response to a change in a Java resource.

Any extension to IDEs should strive to keep the immediacy provided by this fast build process. This can be achieved by making the extension part of the incremental build process and delivering its results fast enough to keep the workflow of the developer using the IDE uninterrupted. To reach this goal, the static analyses themselves have to work in an incremental fashion.

Most available static analysis tools have the following properties that make them less suitable for integration into an IDE:

- They are implemented as monolithic tools, with a standalone user interface which makes it difficult to integrate their results into an IDE.
- No incremental build integration is available and retrofitting it is difficult, because the analyses are written without incrementalization in mind.

The goal of this thesis is to tackle these problems by modeling analyses as small producer–consumer units that can share results, are written for incremental usage and are tightly integrated into an IDE. Making analyses modular removes the overhead of computing base analyses like whole program call graph analyses multiple times. Results of base analyses like the whole program call graph can be used as input for multiple other analyses. This obviates the need for these analyses to recompute the call graph from scratch.

1.2 Thesis

Incrementalizing static analyses improves development environments by enabling more and more complex analyses to run alongside the incremental build process.

This thesis focuses on approaches to write static analyses for incremental usage. Currently these analyses are developed in an ad hoc fashion, choosing the approach the developer of the analysis is most familiar with. If the approach taken is not the best for the problem at hand, then either the development will take longer or the analysis will run slower than necessary.

In this thesis, the following approaches for incrementalizing static analyses are analyzed, explored and compared:

Manual incrementalization A general purpose language, such as Java, is used to implement a static analysis that achieves the incrementalization using a special purpose algorithm.

Automatic incrementalization The analysis is written with the full build in mind and is incrementalized automatically. This requires the underlying mechanisms of the language or framework to have a built–in mechanism that can reconcile the results for the changed program.

Means are needed, that allow developers to choose the approach best suited for the analysis. Therefore, the effect of analysis properties on the suitability of the respective approach to incrementalization is investigated. Three static analyses have been implemented using the mentioned approaches. The selected analyses represent different categories of static analyses:

1. **Confined types** represent data flow analyses. The analysis implements a machine checkable programming discipline and prevents leaks of sensitive object references. Confined types originally were developed by Vitek and Bokowski [135] to enforce security properties in Java programs. They have since been used to enforce domain specific coding restrictions for enterprise java beans [35]. A formalized version [138] was used to reason about safety properties in a JVM-like environment [66].

   The confined types analysis is an example for an optional type system [22]. *Optional type systems* are type systems that do not influence the runtime semantics, but flag certain kinds of errors at compile time. This allows multiple type systems to coexist and to be checked by static analyses. For the implemented analysis, the additional type rules coexist with the built-in Java type system. The violations of the optional

type system are reported by the static analysis and presented to the developer together with the warnings and errors of the Java type system as reported by the compiler.

2. The **incremental rapid type analysis** represents control flow analyses. The Rapid Type Analysis (RTA) constructs an interprocedural call graph analysis that omits calls to methods in types that are never instantiated. It is an infrastructural analyses that provides its results for further analyses.

3. The **ensemble based architecture enforcement** is an example for analyses that check structural properties of programs. The analysis checks for deviations between a specified structure of the program and the implemented structure in terms of uses between source elements. An example for a specified constraint is that only factory classes should access constructors of product classes when the factory pattern [67] is employed. Unintended uses are considered violations of the specified structure and therefore treated as errors.

The following properties of the analyses are examined:

Modularity with respect to input: This relates the granularity of input changes to the granularity of output changes. If, for example, an analysis is modular on class level, the changes in a class that is read as input relate to changes in a corresponding structure for the output, but does not change output that correspond to other classes.

Expressiveness of configuration languages: Configuration mechanisms are necessary to enable the use of analyses with different programs. It is necessary to, e.g., signify the start methods for a call graph construction analysis. The configuration mechanisms may be expressed in different forms (e.g. APIs, configuration files, program snippets) but can be seen as configuration languages. For incremental analyses, changes to configurations written in these configuration languages have to be evaluated. With increasing expressiveness[2] of the configuration language, the implementation of this incremental evaluation gets harder. A simple configuration language could, for example, enumerate source elements (e.g. classes or methods). A configuration language that is more expressive could provide complex queries to select groups of source elements.

[2] A language construct is *expressive* [64] in a language, if its translation to the remaining language enforces a global reorganization of the entire program.

Modularity of data structures: Are data-structures necessary that accumulate knowledge about the whole program at once, or does local reasoning suffice? Again, for incremental analyses, changes to its input need to be evaluated. For internal data structures that are influenced through a limited subset of input, it may be practical to delete and recompute affected data. This is not the case for data structures representing facts about the whole program. These need to be incrementally maintained, which is time consuming and error prone to develop.

The analyses are implemented twice; once using automatic incrementalization and once using manual incrementalization. The implementations are compared according to the following criteria:

- running time of the analyses (measured in seconds)
- development time for the analyses (measured in developer–days)
- implementation size of the analyses (measured in lines of code)

Based on these measurements and the properties of the analyses, conclusions are drawn, how the approaches compare and which approach is favored by which properties.

1.3 Contributions

This thesis makes the following contributions:

- An approach to model analyses as modular producer-consumer units is presented that includes means to support the incrementalization of the analyses. Magellan, an implementation of the approach is described and its integration into the incremental build process of Eclipse is discussed.
- The use of automatic incrementalization for analyses of Java code is integrated into Magellan and the incremental build process of Eclipse. The approach is contrasted with the manual incrementalization.
- Means are developed, guiding a decision whether to use a framework for automatic incrementalization or to manually incrementalize a static analyses under development. It is shown, that the expressiveness of the configuration language for the analysis has important consequences for suitable incrementalization approaches. Incremental static analyses that incorporate intensional, query-based configuration languages

should be implemented using environments that support automatic incrementalization. For incremental static analyses, where the domain knowledge allows for domain specific optimizations, which are difficult to integrate into a framework for automatic incrementalization, the manual approach is better suited.

- An incremental version of the confined types analysis, originally developed by Vitek and Bokowski [135] is presented.
- An incremental version of the rapid type analysis, originally developed by Bacon and Sweeney [12] is presented. The analysis is extended to work with many features of the full Java language.
- An incremental analysis that checks constraints on the dependencies of groups of source elements is presented. Declarative queries are used to group source elements uses into so called *ensembles*. These ensembles may overlap and may reach across programming language module boundaries such as classes and packages. The analysis uses a domain-specific language that also supports parameterized constraint templates, which can be re–used for expressing several instances of a certain constraint type.

  A visual notation is proposed for the comprehensive specification of high–level architectural dependencies; its constructs are implemented in terms of the core logic-based language. Meta–data attached to source code elements is used in template constraints to define dependency constraints on ensembles representing roles in design patterns.

In the framework of the research done in this thesis the following papers have been published:

1. M. Eichberg, S. Kloppenburg, M. Mezini, and T. Schuh. Incremental confined types analysis. In *Proceedings of the Workshop on Language Descriptions, Tools and Applications 2006*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.
2. M. Eichberg, M. Mezini, S. Kloppenburg, K. Ostermann, and B. Rank. Integrating and scheduling an open set of static analyses. In *Proceedings of the International Conference on Automated Software Engineering 2006*. IEEE Computer Society, 2006.
3. M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. *Proceedings of International Conference on Software Engineering 2008*, ACM, 2008.

1.4 Organization of the Dissertation

The remainder of thesis is organized into the following chapters:

- Approaches for the development of incremental static analyses are discussed in Chapter 2. Manual incrementalization is introduced first. The state of the art in analysis integrated IDEs is introduced by presenting Magellan, an open platform for static analyses. The approach for automatic incrementalization is described next. After a short introduction to Datalog, the approach and environment used for automatic incrementalization are discussed.

- Chapters 3 to 5 present the analyses that are selected to compare the approaches to incrementalization. In each chapter, first the analysis is discussed. Then, the implementations according to the automatic and the manual approach are presented and compared to each other. The analyses are described in the following chapters:

  - The approach to incrementally check for violations of the optional type system for confined types is described in Chapter 3.
  - The incremental approach to rapid type analysis is presented in Chapter 4.
  - The ensemble based architecture analysis is an incremental static analysis for concern modeling and structural dependency checking and is discussed in Chapter 5.

- In Chapter 6 the conclusions drawn from the work are presented and options for future work are discussed.

Chapter 2

Incrementalization of Static Analyses

Static analysis is the analysis of computer programs that is performed without executing the analysed program. The input to the analysis is a program representation generated from the source or object code of the program. The analysis itself is done by an automated tool.

This chapter discusses approaches to achieve incrementalization of static analyses. The next section introduces the need for incremental static analysis as part of IDEs. Section 2.2 presents means to support development of manually incrementalized static analyses. Section 2.3 discusses an environment that supports automatic incrementalization of static analyses. In Section 2.4 criteria for comparing the approaches are presented. Section 2.5 summarizes the chapter.

2.1 Introduction

Static analyses commonly are used as part of compilers to optimize and check code for possible errors. Also, many standalone tools exist that use static analyses to check code for properties, such as bug patterns, coding conventions or security properties.

The following problems prohibit widespread use of standalone static analysis tools:

- Each tool has its own user interface and therefore has to be newly learned.
- To run the analysis a context switch away from the development environment is necessary.

- Analysis runtimes usually are quite long. Running the analyses is therefore often delayed until late in the project or at most integrated into nightly builds and not run on demand nor as part of the development process.

Integrating these analyses into development environments and especially into the incremental build process offered by modern IDEs brings several benefits:

- Tool adoption issues are reduced, as the user interface of the IDE is re–used.
- Less context switches for the developer are necessary, as the result of the analyses can be presented to him together with the compiler messages.
- The results are immediately available to the developer enabling her to fix errors during development, when she is still aware of the context of the error.

These benefits are also valid, if the analyses are integrated into the compiler, and the user interface of the IDE is used to display the results. Yet, many analyses are domain specific (for example, to check the coding conventions for Java EE applications), and therefore are not suitable for inclusion into general purpose compilers. Also, compilers for languages that are in widespread use (such as Java), are not easily extended by third party developers. IDEs on the other hand offer plugin infrastructures that allow third party extension.

The third problem mentioned above is the long analysis runtime. Performance is especially crucial for an integration into the incremental build process. The time required to run analyses is acceptable as long as the incremental build is finished when the developer tries to perform the next save operation. When the build is still running, storing the changed file has to be postponed until the build process has finished to avoid an inconsistent state of the database. Until then, the developer has to wait and cannot continue editing the code. Hence, the analysis has to be fast enough that this situation does not occur frequently in practice.

Redoing analyses for the whole program can be quite time consuming, which makes integrating these analyses into an incremental build process of an IDE a challenge as developers expect immediate feedback from their IDE. Compilers face the same problem. Therefore, state of the art compilers work incrementally and recompile only changed files and files depending on the changed files. This speeds up the compilation process by several orders of magnitude.

Thus, if static analyses are to be integrated into the incremental build process, they need to be changed to take the incremental changes of the program and the analysis result from the last run as additional input. From this change, similar speedups are to be expected.

Incremental static analysis first appeared as part of compiler construction [41, 115] and is still of interest there (e. g. [29]). Outside of compilers, build tools make use of similar concepts. Ant [137] and Gnu Make [63] use statically generated dependency files to selectively recompile only changed files and their dependencies.

Integrating incremental static analysis into IDEs and especially the incremental build process offers the following additional benefits:

- The incremental build process provides a set of added, changed and removed artifacts, so the analysis can be notified of these changes.
- The set of error messages can be updated incrementally.

The static analysis platform that in this thesis is used as foundation to study the incrementalization of static analysis, itself is based on the Eclipse IDE. Eclipse[1] is a general purpose open–source platform. Since version 3.0, Eclipse itself contains only a kernel with plug–in loading capabilities. All additional functionality is provided by plug–ins. Eclipse serves as a basis for various *Rich–Client applications*, as well as IDE for various programming languages. Common examples include Eclipse CDT for C/C++, Eclipse PDT for PHP and Eclipse COBOL for COBOL. Most important for this work is the Eclipse IDE for Java, which is realized by the *Java Development Tools* project (JDT). The JDT is a set of plug–ins for Eclipse that extend the Eclipse framework to a full–featured Java IDE, which contains, among other features, an incremental builder, error reporting, debugging support, code completion and syntax highlighting.

There are two basic types of builds in Eclipse:

- Full builds perform a build from scratch. They treat all resources as new. All artifacts generated in previous builds are removed from the workspace, and a build process is started for all source artifacts.
- For incremental builds, the previous build state is remembered by the builder. This leads to an optimized build based on the changes in the project since the last build. Incremental builds are seeded with a resource change delta. The delta comprises the effect of all resource

[1] `htt://www.eclipse.org`

> changes since the builder last built the project. For example, the internal state maintained by the Java builder includes things like a dependency graph and a list of compilation problems reported. This information is used during incremental builds to identify which classes need to be recompiled in response to a change in a Java resource.

By choosing Eclipse as the underlying framework many issues related to tool adoption [15, 61] are already solved. By building on top of the incremental build process, the user will perceive no difference between the checks carried out by the standard Java compiler and the analysis. This flattens the learning curve, as it is not necessary to learn how to use the tool, provided the developer is already familiar with Eclipse. Additionally, as far as the standard Eclipse views are (re)used for configuration and to visualize feedback, no user interface related issues arise.

2.2 Manual Incrementalization

This section shares some material with Integrating and Scheduling an Open Set of Static Analyses [55].

To manually incrementalize a static analysis, a developer uses a general purpose programming language, such as Java, to develop an analysis that works in an incremental fashion. Usually the development of incremental static analyses is based on existing analyses that work in a non–incremental fashion and recompute their result for each analysis run from scratch.

The difficulty of incrementalizing static analyses varies with the amount of amount of code that influences a given subset of the analysis' output. The smallest amount of code is a single statement of the program under inspection. An example is an analysis that checks for assignments that are part of test expressions in if–statements. The largest amount of code is the whole program, for example to compute the reachability of methods.

If the underlying non–incremental analysis works in a modular fashion, which means, that the analysis examines parts of the input on its own, without creating data structures concerning the program as a whole (so called *whole program facts*, then the following approach can be taken: In a first, full build, that analysis proceeds as in the non–incremental version and analyses the whole program. For changes to the program, all results for removed parts of the program are removed from the analysis, and the added parts of the program are analysed and the results are added to the analysis results.

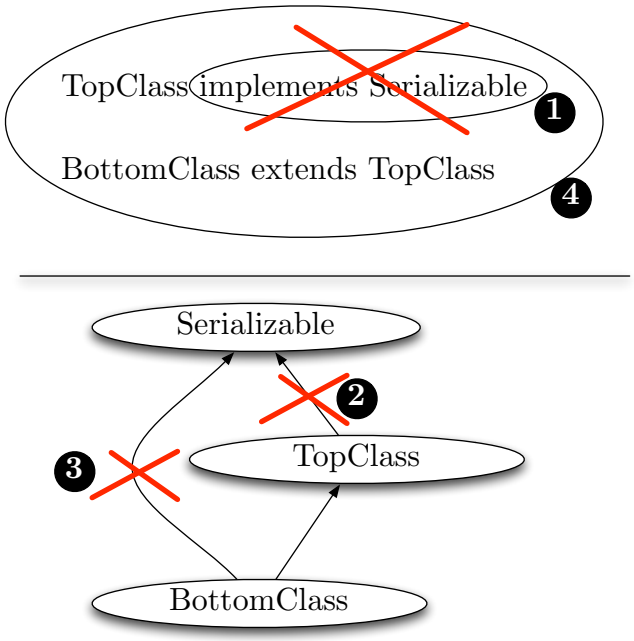


Figure 2.1: Calculating the Extent

If the underlying non–incremental analysis creates whole program facts, then the approach is extended by calculating the extent of the changes and then removing the invalidated parts of the result.

The *extent* of a change to a program with respect to an analysis result, is the subset of the analysis result that is no longer correct after the change. For example: A type hierarchy analysis is done for a Java program, where one of the top classes (TopClass) of a hierarchy inherits from java.io.Serializable . Then this class is changed in such a way, that it no longer inherits from Serializable. The extent of this change is not only the class itself, but also all classes that inherit from the class. Theses classes also no longer implement Serializable and thus, have a changed type hierarchy.

In general, the extent is the transitive closure of the image of the change. Figure 2.1 visualizes the example. The change (number 1) is, that TopClass no longer inherits Serializable. The image of this change (number 2) is, that the type hierarchy for TopClass no longer contains Serializable as implemented interface. This propagates (number 3) to the type hierarchy entries of all classes inheriting from TopClass (the transitive closure of the change).

The way to compute the extent differs from analysis to analysis and may be quite complicated. Once the extent is computed, the subset of the program that led to the analysis results (called *pre–image of the extent)* needs to be

calculated. This can be done by annotating the analysis results with the program elements that caused the result.

The pre–image of the extent is the set comprising TopClass and all classes inheriting from TopClass (number 4). So, if the type hierarchy has to be updated after a change to the inheritance relationship of TopClass, it does not suffice to re–analyse TopClass. BottomClass needs to be re–analysed too, although its source code did not change.

Then the extent has to be removed from the analysis result and its pre–image (that comprise changed code and its dependents) have to be re–analyzed together with added code.

In addition to changes of the program under inspection, changes to configurations of the analysis have to be taken into account. Static analyses need to be parameterized to allow their use with different programs. It is necessary to, e.g., be able to configure the start methods for a call graph construction analysis. These configuration mechanisms appear in different forms; common possibilities comprise APIs, configuration files or program snippets that are customized. All these mechanisms can be seen as configuration languages.

For incremental analyses, changes to configurations written in these configuration languages have to be evaluated. The difficulty of writing software for this purpose increases with the expressiveness of the configuration language. A language construct is called *expressive* [64] in a language, if its translation to the remaining language enforces a global reorganization of the entire program. An example for a configuration language with little expressiveness is a language, that enumerates source elements (e.g. classes or methods). An example for configuration language that with more expressiveness is a language that provides means to formulate complex, intensional queries to select groups of source elements.

To allow static analyses developed by independent developers to work together and to maximize reuse of analysis results as well as development effort, a common, open platform for static analyses is needed. This platform should provide the following features:

- Means for coordination of the analyses, as common analyses, such as call graphs, should be reusable across analyses.
- Means for parallelization of analyses, as multi core architectures become the norm.
- Interfaces handling deltas of analysis results to enable incremental analyses the cooperation with each other thus avoiding unnecessary re–computation of deltas and analysis results.

The following sections introduce a platform that provides these features. The next section formulates the need for an open platform for static analyses. Section 2.2.1 introduces Magellan, which implements such a platform. Section 2.2.3 details the specification of analysis dependencies in term of their input and output data. Section 2.2.4 presents related work.

2.2.1 IDE–Integrated Platforms for Static Analysis

As already stated, static analyses are used to check that certain desired properties hold before executing a program. It should be possible to use these analyses only when needed, because otherwise, CPU–time and memory is wasted to check for constraints in one domain, when the program under development belongs to a different domain. It should also be possible to easily extend the set of analyses. Static analysis tools that support only a fixed set of analyses [8, 59, 93, 102] are not well–suited for project–specific analyses. Other tools [52, 82, 96, 114] provide a meta–programming API (or language) which can be used to implement and integrate new analyses.

A mechanism is needed to provide coordination for sets of analyses that depend on the results of each other; in existing tools this has to be done manually, if it is possible at all, which makes it hard to integrate a sophisticated net of interdependent analyses. As a result, these tools are usually only extended with analyses that do not depend on the results of other analyses.

Most tools for static analysis are monolithic standalone tools. This has several major drawbacks:

duplicate work: Common functionality, such as creating a suitable code representation, or creating a call graph is duplicated in each tool, because the work done for other tools can not be reused.

unnecessary context switches: Because the tools are not integrated into the IDE, developers need to switch contexts form development to analysis tools.

error reporting: As the tools do not run inside the IDE, the developer has to map the error message manually to the code before being able to fix the reported error.

To remedy the first shortcoming, an open platform allows to add or remove analyses as needed. The developer is allowed to select a subset of the available analyses. To improve on the second and third point, a tight integration of the platform into the incremental build process of an IDE is necessary. The analyses run along the incremental build process of the IDE.

As a result, the developer receives immediate feedback on the effect of source code changes. As even a small change, e.g., to the type hierarchy, may cause drastic changes to previous analysis results, immediate feedback is important. Otherwise, the developers will continue editing the source code using outdated analysis results. Tracing to the root of changes in the analysis results only after the next full build is time consuming; immediate feedback is much more effective.

Some tools, such as PMD [76], that started out as standalone tools, developed integration with IDEs. PMD has front ends for JDeveloper, Eclipse, JEdit, JBuilder, BlueJ, CodeGuide, NetBeans / Sun Java Studio Enterprise / Creator, IntelliJ IDEA, TextPad, Maven, Ant, Gel, JCreator, and Emacs. Other tools specialize on one IDE, e.g. Jackpot[2] is a Netbeans module to support reengineering of Java Source Code. Reengineering is a super–set of refactoring that includes API Migration, redesign and anti–pattern correction. Jackpot is a rule engine that transforms the result of custom queries over the AST of the project. The query language, designed by James Gosling, matches patterns on the AST, filter them with conditions and transforms them. For example, the rule

```
$object.show() => $object.setVisible(true) ::
      $object instanceof java.awt.Component;
```

converts any statement which invokes the deprecated Component.show() method to Component.setVisible(**true**), but only when the object's class is derived from Component.

IntelliJ IDEA[3] includes a dependency structure matrix (DSM) module[4], that display dependencies between packages or classes. It can check for dependency cycles and includes source code navigation. This is one of the many code inspection modules[5] that IntelliJ IDEA provides. As IntelliJ does not intend third parties to extend the set of inspection modules, there is no public API, nor the possibility to build upon the work done.

Tools that offer support for a multitude of IDEs can only use the common denominator of the supported platforms, whereas tools that focus on supporting one platform exclusively can reuse everything the platform provides.

Allowing third party plugins brings new possibilities, such as reducing the engineering effort for developing new analyses and to support more efficient

[2] http://jackpot.netbeans.org/

[3] http://www.jetbrains.com/idea

[4] http://www.jetbrains.com/idea/features/dependency_analysis.html

[5] http://www.jetbrains.com/idea/documentation/inspections.jsp

use of computational resources needed to execute the analyses. Speeding up the execution is an important prerequisite for integration into the incremental build.

Opening up to third party plugins also brings new, interesting problems, such as defining useful interfaces for the analyses and scheduling an open set of analyses.

2.2.2 Magellan

Magellan is a framework for coordinating and scheduling static analysis that is tightly integrated with the Eclipse IDE. Magellan has an open data model to store the results of analyses, which allows the integration of analyses developed by third parties.

Magellan is realized as a bundle of Eclipse plugins and coordinates analyses written in Java and allows the embedding of external query engines. XQuery [19] and XSB [118] are two engines that already are embedded into Magellan. The configuration of the analyses is done via the *MagellanUI*. This configuration is stored together with the set of available analyses and their properties in the *AnalysisRegistry*. When the configuration is completed, the *Scheduler* accesses the AnalysisRegistry to get the set of user–selected analyses and generates a schedule for the configuration. This schedule is passed to the *Dispatcher*, which is registered with the Eclipse build system and calls the analyses in the appropriate order. Each analysis then accesses the WPDB to get its input and to store its output. The *WPDB* (the *whole program data base*) is the data store for all analyses. Analyses may use the *ProblemsView* to inform the developer of its results.

This requires means of coordination between analyses that write and read the data model.

Table 2.1 on the next page illustrates that static analyses differ widely in the data they require, but also share subsets of data. For example, both the SA and the CFT checker require data flow information. Each analysis could of course compute all the data it requires from the raw source code or from a generic representation of the project. However, implementing and running several instances of an algorithm for data flow analysis wastes both engineering effort and computational resources. Furthermore, analyses may consume only information about a part of the project. For example, the EH analysis requires only information about the interfaces of Java classes; method bodies or other artifacts such as deployment descriptors are irrelevant. Hence, it is a waste of resources to reify a generic representation of the entire software.

To cope with the issues stated in the previous paragraph, it is desirable to divide the analyses into small modular producer–consumer units. Analyses

ID	Description	Required Data
NSF	Searches for finalize methods that do not call **super**.finalize.	control flow graph (CFG)
EH	Searches for Java classes overriding either equals(**boolean**) or hashCode(), but not both.	interfaces of Java classes
SA	Searches for String.append(..) invocations where the return value is ignored.	data flow information
CTAV	Searches for Enterprise Java Beans that use declarative and programmatic transaction demarcation [42].	type hierarchy, method bodies, EJB deployment descriptors
CFT	Realization of Confined Types [54] based on Java annotations.	type hierarchy, type hierarchy changes, data flow information, public interfaces of libraries

Table 2.1: Sample analyses and the data they depend on

such as SA and CFT can share the results produced by a base analysis for data flow information; similarly, EH can consume the results of an analysis that produces information about the interfaces of Java classes only. This requires that analyses are run in a well defined order to satisfy their data–producer–consumer relations.

These relations cannot, however, be expressed by a predefined total order, since the set of analyses is open and any number of—as yet unknown—analyses could be required to be scheduled before a specific analysis. The producer–consumer dependencies cannot be represented by a partial order graph either. For better performance, some analyses should be able to transform and modify existing analysis data instead of generating new data. Furthermore, several analyses that generate the same information can co–exist within the platform and it should be ensured that at most one of them is run. Both cases are not expressible by a partial order. Last but not least, to leverage modern multi–processor architectures, it is also desirable to parallelize analysis executions whenever possible.

It is also desirable to automatically select and run only analyses that produce information consumed by analyses directly selected by the user. End–users, in general, select only a subset of all available analyses; therefore it is desirable to automatically select and run only the minimum set of analyses that produce information consumed by the selected analyses. A base analy-

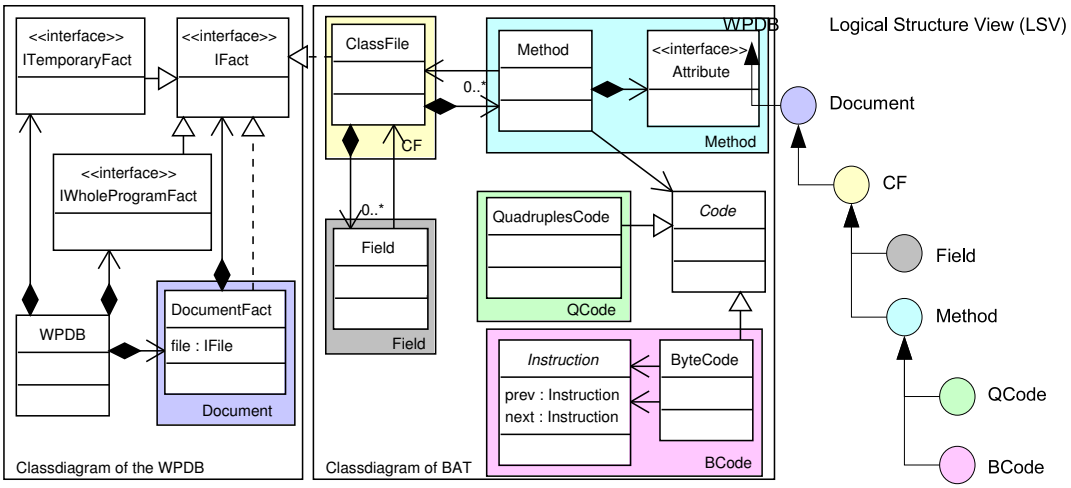


Figure 2.2: A part of the LSV and its mapping to the WPDB

sis, e.g., for getting the type hierarchy, should only run if its result is needed by a user selected analysis.

Hard–coding all the dependencies and execution order into the analyses themselves would prohibit the extensibility of the platform. Manually scheduling the analyses for a given configuration of the platform is also very cumbersome. Hence, an automated approach to scheduling analyses is required.

Magellan coordinates analyses based on solving constraint systems that represent the dependencies between the analyses. The coordination unit, called *scheduler*, treats analyses as modules that write, read or maintain parts of the open data model. Each analysis describes its properties and dependencies in a special analysis specification language (ASL). These specifications are mapped onto a constraint system which is fed to a constraint solver. To calculate a schedule that is optimal with regard to the number of internal analyses to run and the parallelization of the analyses to be executed, corresponding objective functions are added to the set of constraints.

The Analysis Data Model

The analysis data is stored in the *whole-program database* (WPDB). The WPDB is an object graph built–up cooperatively by the executed analyses. The WPDB has a set of designated root objects which are called *facts*. The architecture of the fact objects is shown within the box on the left–hand side of Figure 2.2, entitled “Class diagram of the WPDB”. There are three different types of facts.

For each resource (file) in the project a *document fact* is created (an object of class `DocumentFact` in Figure 2.2 on the previous page), which keeps a reference to the underlying file. A document fact contains a set of facts, represented by implementations of the `IFact` interface. Analyses can attach derived information about the resource to its set of facts. A representation of a Java class file is a typical example of a fact aggregated within a document fact. Instances of the class `ClassFile`—within the box in the middle of Figure 2.2 on the preceding page—represent individual Java class files produced by the Java Bytecode Analysis Toolkit BAT [50].

A document fact is automatically created, added to, or removed from the database corresponding to the type of action on the underlying file. The set of all document facts that are created or removed from the database in a build is also directly made available to the analyses. This enables analyses which can perform their work incrementally per document to process only the delta to the previous build.

Information that cannot directly be associated with specific documents is stored in the database using *whole program facts*. A whole program fact always needs to be maintained by the analysis that creates it. After a full build, the analysis has to re–create the whole program fact; after an incremental build, the analysis has to bring the information up–to–date to reflect the current project's state.

For example, an analysis that makes the type hierarchy information available has to update the type hierarchy whenever the developer makes a change that invalidates the "old" type hierarchy. Information that is only valid during a build step is stored in *temporary facts*. All temporary facts are automatically deleted before each build. For example, a type hierarchy analysis could also make information about the changes to the type hierarchy available for the benefit of subsequent analyses. However, this information is only valid for the current build.

Data dependencies in the WPDB are expressed in the *logical structure view* (LSV). The logical structure view is a directed acyclic graph. Every node in the LSV stands for a part of the WPDB, whereby a part of the WPDB can be a selection of objects or, even more fine–grained, a selection of field values of the objects in the WPDB. The nodes in the LSV are called *entities*. Figure 2.2 on the previous page shows a part of the LSV on the right–hand side. Also, its mapping to the corresponding parts of the WPDB is shown by the gray boxes around elements of the WPDB and BAT class diagrams. Consider for an example the gray box labeled "Method" surrounding the class `Method` and `Attribute` in the BAT class diagram. This boxing states that a LSV method entity is mapped to a WPDB method and all its attributes. Entities in the LSV can be referred to by using paths in the LSV starting

at the WPDB vertex; e.g., the following path refers to the BCode entity: **Document**/CF/Method/BCode.

Edges in the LSV express data dependencies as follows: *If data in the WPDB is changed that belongs to an LSV entity v, then all data in the WPDB that is invalidated by the change is associated to entities w such that there is a path from w to v in the LSV.* Declaring an entity w as dependent on an entity v implies no conflict between an analysis that changes the data associated to w or any of its dependent entities and those that just read the data associated to v. Further, analyses that access siblings do not conflict. For example, `Field` and `Method` are declared as dependent entities of `CF`. Hence, an invalidation of the information on a class entity automatically invalidates information on its fields and methods. But, there are no conflicts between analyses that process `Field` and `Method` entities respectively. These properties are leveraged by the scheduler to parallelize analysis executions. Though a fined–grained LSV increases the possibilities for parallelization, it decreases the ease of describing and understanding the dependencies among analysis data.

The LSV is derived from the set of analysis specifications. The mapping between the LSV and the WPDB is specified informally in the documentation of the respective WPDB elements.

If the user of the platform would like to extend the predefined LSV and WPDB, for example to make the intra–procedural control–dependence graphs (CDG) of methods available, he first needs to determine where to store the information. The representation for class files enables extension of its object graph by means of attributes. Hence, the user could implement a set of classes for managing the CDG and store instances of them as attributes of the corresponding code object. Since the CDG is derived from the code of the method, the LSV is extended with a new node CDG which is associated with all CDG objects in the WPDB, and an edge to, e.g., BCode to represent the dependency.

2.2.3 Specifications of Analyses

The *analysis specification language* (ASL) is used to declare the *data* required and provided by each analysis in terms of the logical structure view described in the previous section. The ASL supports six different types of dependencies as shown in the ASL grammar in Figure 2.3 on the following page. Listings 2.1, 2.2 and 2.3 on the following pages illustrate the specification of the sample analyses from Table 2.1 on page 36.

A **reads** dependency on some LSV entities means that the analysis works incrementally on the specified input data. For example, the EH checker

AS	::= **analysis** ID STATEMENT*
STATEMENT	::= DEPENDENCY PATH*
DEPENDENCY	::= **reads-global** | **reads** | **writes** | **invalidates** | **maintains** | **writes-temporary**
PATH	::= ID [/ PATH]

Figure 2.3: The ASL grammar

(Listing 2.2 on the facing page, Line 2) specifies that the analysis will read the entities referred to by the path expression **Document**/CF/Method. A **reads −global** dependency, on the other hand, means that the analysis needs data of the specified kind for *all* documents, not just those processed in the current build. The current implementation of the type hierarchy analysis, e.g., needs access to all class files, not just those changed; hence, the corresponding **reads−global** dependency in Listing 2.1 on the next page, Line 11.

A **writes** dependency specifies that the analysis provides data of the specified type for documents that are changed in the current build step only. For example, the DDP analysis (Listing 2.3 on page 42, Line 6) specifies that it writes the EJBDD entity and implicitly reads the preceding entities, i.e. the **Document** entity. If all path elements would be considered as written it would not be possible to have a second analysis that **writes** a dependent entity, but which does not write the preceding entities; e.g., it would not be possible to specify that an analysis just writes a BCode's CFG and not the BCode. A **writes−temporary** dependency is used for data that is automatically invalidated (and hence removed by the platform) before the next build. For example, the type hierarchy analysis (Listing 2.1 on the next page, Line 10) also provides information about changes to the type hierarchy between the current and the previous build. Since this information is only valid for one specific build step, it is declared using **writes−temporary**. As in case of **writes**, only the last entity of the path is written and the previous entities are read.

The **invalidates** dependency specifies that after executing the analysis the last entity referred to by the given path expression is no longer valid. This is usually the case if an analysis provides its result by transforming existing data in the WPDB. For example, the analysis which transforms a method's bytecode representation into the 3–address based representation (Listing 2.1 on the facing page, Line 2) specifies that the BCode entity will become invalid when the analysis is executed because the analysis changes the existing data in the WPDB.

Finally, **maintains** is used by an analysis to declare that it creates an entity and updates it during the following builds. For example, the type hierarchy

```
1  analysis BCFG writes Document/CF/Method/BCode/CFG (* creates the
       control-flow graph (CFG) *)
2  analysis BtoQ (* transforms the Bytecode in 3-address SSA form *)
3    reads Document/CF/Method/BCode
4    invalidates Document/CF/Method/BCode
5    writes Document/CF/Method/QCodeSSA
6  analysis LIB (* maintains the repository of used library classes *)
7    reads Document/CF/Method/BCode
8    reads-global Document/CF
9    maintains Library/CF/Field_NON_PRIVATE, Library/CF/
         Method_NON_PRIVATE
10 analysis TH (* maintains the type hierarchy *)
11   reads-global Document/CF, Library/CF
12   writes-temporary TypeHierarchyChange
13   maintains TypeHierarchy
14 analysis CTA1 (* programmatic and declarative transaction demarcation is
       used *)
15   reads Document/EJBDD
16   reads-global TypeHierarchy, Document/CF/Method/BCode
17   writes CTAViolations
18 analysis CTA2 (* alternative CTA analysis *)
19   reads Document/EJBDD
20   reads-global TypeHierarchy, Document/CF/Method/QCodeSSA
21   writes CTAViolations
```

Listing 2.1: Base Analyses that read, create and transform the database

```
1  analysis NSF reads Document/CF/Method/QCode/CFG (* finalize does not
       call super.finalize() *)
2  analysis EH reads Document/CF/Method (* equals and hashcode have to be
       implemented pairwise *)
3  analysis SA reads Document/CF/Method/QCodeSSA (* String.Append()
       must not be ignored *)
4  analysis CFT (* realizes Confined Types *)
5    reads TypeHierarchyChange
6    reads-global TypeHierarchy, Document/CF/Method/QCodeSSA,
7            Library/CF/Method_NON_PRIVATE
8  analysis CTAV reads CTAViolations (* wraps CTA and CTA2 *)
```

Listing 2.2: Analyses that just read the database (Checkers)

```
1 analysis CFP (* creates class file representation *)
2   writes Document/CF, \
3                  Document/CF/Field, \
4                  Document/CF/Method, \
5                  Document/CF/Method/BCode
6 analysis DDP writes Document/EJBDD (* creates EJB Deployment
    Descriptor representation *)
```

Listing 2.3: Analyses that make the base representations available

analysis declares to maintain (Listing 2.1 on the previous page, Line 13) the TypeHierarchy entity.

Analyses may overlap in both their input and output data. If multiple analyses are present that can produce the same data, the scheduler decides which of these analyses will be executed. There can also be multiple analysis specifications for the same analysis. This can be used to express that an analysis needs one entity *or* another kind of entity. For example, the checker for detecting conflicting transaction demarcations (CTAV – Listing 2.2 on the preceding page, Line 8) needs either the byte code (BCode – Listing 2.1 on the previous page, Line 14) or the SSA–transformed code (QCodeSSA – Listing 2.1 on the preceding page, Line 18), hence there are two analysis specifications for this analysis. Such alternatives give the scheduler more leeway in scheduling an analysis.

An analysis specification also serves as a contract on what the analysis implementation is allowed to do with the WPDB. The result of an analysis must only depend on data in the WPDB whose entity in the LSV is read. The analysis must not add any data to WPDB entities which are not marked as **writes** or **writes–temporary** nor change any data that is not marked as **invalidates** or **maintains**, respectively. The schedule computed by the scheduler is correct if and only if all analyses are correct w.r.t. their specification and if the LSV correctly models the dependency relations in the WPDB.

2.2.4 Related Work

Several extensible tools for analyzing software projects have been developed. These tools can be divided in two broad categories. In the first category, there are tools that enable the developer to implement new analyses using declarative query languages. For instance, PQL [96] is a specially developed query language, CodeQuest [73] uses Datalog [132], XIRC [52] uses XQuery [19] and Xgcc [9, 60, 75] uses its own state–machine based language Metal. The second category consists of tools that provide an API for developing analyses,

such as IRC [53], FindBugs [82] or PMD [38, 76].

The tools of the first category have in common that the information that is made available about the programs is fixed. Analyses are strictly divided into two categories.

1. Tool internal analyses to build up the information about the project.

2. User defined queries executed in a second step.

In PQL [96], for example, the source of information is a context–sensitive, flow–insensitive, inclusion–based pointer alias analysis. However, the analyses that create the program database are executed independently of the needs of the actual queries.

Further, since the set of analyses that make up the program database is fixed, these tools are targeted toward a specific type of analysis. For example, PQL [96] is particularly well–suited for data–flow related analyses. XIRC, on the other hand, was designed to check structural properties of classes. While being very useful for detecting certain types of errors, and being extensible within a particular problem class, these tools cannot be used as platforms for the implementation of a broad range of analyses. A second consequence of always executing a fixed set of base analyses is that, if sophisticated non–incremental analyses, e.g., as in case of PQL, are executed, the time to update the database is too lengthy to enable an integration with the incremental build process. An advantage of these approaches is that conflicts between analyses that are executed in parallel cannot occur; the queries perform read–only access to the program database. Hence, an explicit scheduling of analyses is not necessary.

Tools of the second category, i.e., tools that provide an explicit API for the development of new analyses, also provide a specific representation of the program's code that is to be used for the implementation of the analyses. For example, FindBugs [82] uses the Java bytecode library BCEL [16] as the basis for the representation of the program's code. BCEL provides an object–oriented representation of a Java class file and implements a basic intra–procedural data–flow analysis. IRC [53] uses an approach comparable to Findbugs; PMD [76] uses the abstract syntax tree. Though, it is technically possible that analyses implemented in Java / C++ that operate on an object graph can refine or transform the graph, these operations are not supported by the frameworks. A transformed representation might conflict with other analyses executed thereafter. However, even when a developer decides to extend a framework's representation by implementing a new analysis that additionally provides a higher–level intermediate representation, the execution of analyses that operate on top of the new intermediate representation is

not supported. The tools do not provide basic functionality for dependency management of analyses. Handling dependencies between analyses is, however, required to ensure the execution of an analysis that provides additional information before the analyses that want to access the information.

Though the proposed approach can also be used to realize a build management tool such as Make [127], this is not in the focus of this work. In case of a static analysis platform the execution of the user selected analyses is the focus. The effect of the analyses on the underlying data is not a concern of the user. In case of a build management tool the user is just interested in getting the result, e.g., the executable, which tasks generated the result is irrelevant. However, when compared with Make the discussed approach provides a more fine grained data model that also enables reasoning about the inner structure of a file. Further, the proposed model also supports the explicit invalidation of entities. In Make every entity is a file and a task must not invalidate (delete) files. Make on the other hand automatically determines which tasks need to be executed to create the new result, whereas Magellan just calls every analysis in case of a change and, basically, each analysis has to determine the scope of entities that need to be processed on its own.

2.3 Automatic Incrementalization

Automatic incrementalization of static analysis is done by engines that maintain a relationship between the extensional facts representing the input of the static analysis and the facts deduced from the intentionally specified output of the analysis. Then, when the extensional fact base (which represents the program under analysis) changes, the corresponding parts of the deduced facts have to be recomputed.

An extended version of a Prolog system for logic programming [118] is a suitable environment and is described next. In Section 2.3.2 the program representation as used in the system is presented. Section 2.3.3 discusses the embedding of XSB into Magellan and Section 2.3.4 describes related work.

2.3.1 XSB Prolog

XSB [118] is a logic programming and deductive database system that provides a Prolog [129] implementation, extended with higher order logic and SLG resolution with explicit negation [5]. SLG resolution is implemented via tabling [43] and ensures, that Datalog programs terminate.

Datalog [31] is a simplified subset of Prolog that will be used for the

```
1 inherits(A,B) :- directInherits(A,B).
2
3 inherits(A,C) :- directInherits(A,B), inherits(B,C).
```

Listing 2.4: Datalog Rule

automatically incrementalized implementations of analyses in this thesis. A Datalog program comprises *facts* and *rules*. Facts are assertions about the represented domain, such as directInherits(object,string) to assert that String directly inherits from Object.

Rules deduce facts from other facts. Rules are represented as Horn clauses as shown in the example in Listing 2.4. The left–hand side of the Datalog clause is called *head*. The right–hand side is called *body*. The head and each of the comma–separated parts of the body are *literals* of the form $\mathsf{p}(t_1,...,t_k)$, where p is a *predicate symbol* and the t_i are *terms*. A term is either a constant or a variable.

To formulate queries against a Datalog program, *goals* are used. A goal is a single literal preceded by a question mark and a dash. In the notation used in this thesis, variables are denoted by beginning with an upper case letter. Constants and predicate symbols are denoted with a string starting with a lower case letter. The goal is fulfilled by the set of facts that match the literal. In the example above, a goal of ?-directInherits(object,A) would return A = string..

If the fact directInherits(string,mystring) is added to the fact base, the goal ?-Inherits(object,A) would return A = string, A = mystring.

Literals, facts or clauses which do not contain variables are called *ground*. In Datalog programs, each fact is ground and each variable that occurs in the head of a rule also occurs in the body of the same rule.

The termination of Datalog programs is ensured by using a purely declarative semantics. Prolog programs on the other hand have an operational semantics, which is defined using the left–recursive depth search in the solution space. If the rule shown in Listing 2.4 was changed to the form shown in Listing 2.5 on the next page, Prolog would never terminate, whereas Datalog will return the same result as above. In XSB, this semantics is implemented via *tabling*. Tabled evaluation [32] is a mechanism to implement declarative semantics as shown above. Calls to tabled sub goals are stored in a table, together with their proven instances. Since consuming sub goals resolve against unique answers rather than repeatedly against program clauses, tabling will terminate whenever a finite number of sub goals are encountered during query evaluation, and each of these sub goals has a finite number of

```
1 inherits(A,B) :− directInherits(A,B).
2
3 inherits(A,C):−inherits(A,B),directInherits(B,C).
```

Listing 2.5: Prolog Recurses Infinitely

answers.

In most implementations of tabling, the table has to be recalculated from scratch if the facts influencing the results in the table change. In the use-case considered in this thesis, namely incremental program analysis, the fact base changes regularly. The corresponding recalculation imposes runtime increases on static analyses using tabling evaluation strategies. To remedy this, incremental tabling was developed [56, 119]. With incremental tabling, only the table entries that depend on changed or deleted facts are removed. Thus, incremental table maintenance as implemented in XSB, makes automatic incrementalization of static analyses feasible.

XSB is embedded into Magellan via interprolog [27, 28], which is a Java interface to a few different Prolog engines. For XSB, two connection modes exist. The stable, and easily debuggable, but slow connection is done via TCP–Sockets. The faster but somewhat less stable connection is done via JNI. Interprolog provides Java representation of Prolog data structures and provides a high–level interface that makes it easy to embed Prolog programs into Java programs.

2.3.2 Representation of Java Programs

To allow an analysis of Java programs in the XSB engine, Java source elements have to be represented as facts. Each type of element is represented using a special fact class. For example, Java method declarations are represented as method/5 facts and Java field as field/4. Each element is associated with an unique id. This allows mix different source elements in analysis results, for example method and type declarations. As an example for the mapping, consider the classes in Listing 2.6 on the facing page showing the class TypeFactory, a factory class that produces flyweights of ObjectType objects; Listing 2.7 on the next page shows the encoding of the class ObjectType.

Type declarations are encoded using type/2[6] (see, e.g., Line 1 in Listing 2.7 on the facing page), where the first argument (or, parameter) is the *id* (identifier) of a source element and the second parameter its type name.

[6]The notation /2 denotes a relation with arity 2.

```
1  package bat.type;
2
3  /* belongs to ensemble: TypesFlyweightFactory */
4  class TypeFactory {
5    ObjectType getObjectType(String fqn) {
6      ObjectType o = pool.get(fqn);
7      if (o == null) {
8        o = new ObjectType(fqn);
9        pool.put(fqn, o);
10     }
11     return o;
12 } }
13
14 class ObjectType extends ReferenceType implements IType {
15   String fqn;
16 /* belongs to ensemble: TypesFlyweightCreation */
17   ObjectType(String fqn) {
18     ...
19 } }
```

Listing 2.6: Example BAT classes

```
1  type(t1, 'bat.type.ObjectType')
2  type(t2, 'java.lang.Object')
3  type(t3, 'java.lang.String')
4  type(t4, 'bat.type.IType')
5  superclass(t1, t2)
6  interface(t1, t4)
7  method(m1, t1, '<init>', [t3], t1)
8  field(f1, t1, 'fqn', t3)
9  type(t5, 'bat.type.TypeFactory'),
10 method(m2, t5, 'getObjectType', [t3], t1)).
```

Listing 2.7: Representation of Java code

Superclass relationships are encoded using superclass/2 (Line 5); the ids of the base type and of the supertype are given as arguments.

Inherited interfaces are encoded by interface/2 (Line 6); the first argument is the id of a type and the second argument the id of the inherited interface. Annotations, thrown exceptions and visibility modifiers are encoded in a similar way.

Method declarations are encoded by method/5 (Line 7) with the following arguments: the method id, the id of the defining class, the method's name, the list of ids of the parameter types, and the return type's id.

Field declarations are encoded using field/4 (Line 8); the arguments encode the field's id, the id of the defining class, the field name, and the type of the field.

Further, the following rules are available, that build upon these relations:

inherits/2 relates the class given as the first argument to all its direct or indirect supertypes.

classesinpackage/2 relates the package name given as the first argument to all classes from that package. Packages do not exist on byte code level and thus have to be reconstructed.

2.3.3 Embedding of XSB into Magellan

XSB is embedded into Magellan via an analysis that stores a whole program fact (`PrologDB`) in Magellan's database. For each full build, a new database is created. The `PrologDB` fact provides methods to interact with the Prolog database via Interprolog. There are method to assert and retract facts, to consult prolog rules and to evaluate queries. Using the assert and retract functionality the analysis maintains the set of Prolog representations of the project's class files.

Consulting rules is done by loading a prolog file with a set of rule definitions. These rules can then be used by other queries or rule definitions. Consulting rules only when required is necessary to avoid the maintenance of tables associated with rules that no user selected analysis uses.

Each query is wrapped by a Java class that is called by Magellan during build processes. When invoked, the Java class passes the query to the Prolog system for evaluation and then processes the result and shows warning messages when appropriate.

2.3.4 Related Work

The first order logic programming language Spine [18], which is similar to Prolog, is used to define statements about Java programs. Design pattern definitions written in Spine are stored in external library files. Given a user request the Hedgehog proof system consults these pattern definitions and checks whether the given Java class(es) meet the pattern definitions. Many of the patterns described in the Design Patterns Book [67] can be expressed but their violation does not necessarily lead to errors.

CodeQuest [73] offers a fast, scalable code querying engine based on Datalog and intended for program understanding and refactoring support. The tool can also be used to check for coding style violations (e.g. public, non–private fields). CodeQuest is slower than XSB, but less memory intensive. As analysis runtime is the main concern, XSB is superior for the chosen application.

The problem of maintaining results incrementally has been analyzed in various fields of research. Surveys are provided by [71] and [97]. Gupta et al [72] presented the counting algorithm and the Delete and Rederive algorithm, which were used as a basis for many other algorithms. Imagine a database query and their solution tuples t, which shall be maintained incrementally. The counting algorithm stores the number of different alternative derivations, count(t), of a tuple t. The influences of a given a change on the number of alternative derivations is calculated and the new count(t) value is determined. A tuple with a count zero is no longer reported as answer of the query. The counting algorithm is limited to non–recursive views.

The Delete and Rederive algorithm first deletes a super–set of the tuples that have to be deleted according to the given change. Then those of them are re–derived, which are still valid.

Based on the idea of Gupta et al [72], Staudt and Jarke [128] present a three–step algorithm similar to the Delete and Rederive algorithm. A declarative program of maintenance rules is derived from the original query definition. This program is then executed to keep the query results up to date. In a second step Staudt and Jarke assume that the query results are maintained externally to point out the usability for client server architectures. A server may not have access to the original materialization (the query results are materialized) but just to the changes when updating the results.

Volz et al [136] extended the algorithm developed by Staudt and Jarke [128] in a way that it is now able to deal with updates and new definitions of queries.

In contrast to the solutions mentioned above the incremental algorithm used in this thesis additionally maintains a dependency structure: The called–

by graph. The incremental updates are not only calculated by executing rules derived from the original query definitions. Instead the called–by graph, which stores information about prior query executions, is used for change propagation. Besides the incremental algorithm can be applied to arbitrary tabled logic programs, especially those that use aggregation and other Prolog built–ins.

2.4 Comparing the Approaches

In this section, criteria for comparing the approaches to incrementalization are presented.

The following theses will be tested:

1. Developing automatically incrementalized static analysis using the XSB environment should be faster than using Java, because Datalog has the benefit, that the programming model is purely declarative and thus more concise and closer to the semantic model of the developer. Developing an algorithm to compute the extent of a change, as it is necessary when developing in Java, can be quite complicated, depending on the analysis in development. Therefore the development time of manually incrementalized analyses is expected to be longer.

2. The runtime of the manually incrementalized static analysis is expected to be shorter that the runtime of the automatically incrementalized version, because the manually incrementalized version can be optimized to the problem at hand. Domain knowledge can be exploited more effectively, as the developer controls a bigger portion of the runtime environment.

3. The effect of these properties of the approaches should vary with the properties of the implemented analysis.

    - Analyses that are modular with respect to their input should be easy to incrementalize using the manual approach, as each module can be analyzed separately.
    - Analyses that heavily rely on whole program facts should be harder to incrementalize using the manual approach, as the extent for changes to the whole program fact needs to be computed.
    - Analyses that incorporate query engines to be used by the developer will benefit most from the use of automatic incrementalization. Analyses that can be configured only in narrow, predictable

ways lend themselves to manual incrementalization as in the domain knowledge allows for domain specific optimizations.

To test these theses, the following selection of static analyses is implemented using both approaches:

Enforcing confined types (Chapter 3) A data flow analysis that implements an optional type system for enforcing security properties. Confined Types are a machine checkable programming discipline that prevents leaks of sensitive object references from their intended domain.

RTA (Chapter 4) A control flow analysis that constructs an interprocedural call graph. Incrementally maintaining the call graph instead of recomputing it from scratch speeds up the analysis time for the call graph significantly. It also eases providing change sets to the call graph, which enables the development of incremental algorithms that build on the call graph.

Ensemble based architecture enforcement (Chapter 5) An analysis that checks for structural properties. It implements an approach for controlling compile time dependencies between groups of source elements. Declarative queries are used to group source elements uses into so called ensembles. Dependencies between program elements can be modeled from different perspectives reflecting architectural, design, and implementation level decisions. Erosion of the intended structure of the code is mitigated by explicitly codifying these different perspectives on the permitted dependencies. Violations are detected continuously and incrementally as software evolves.

The implementations according to the respective incrementalization approaches will be compared along the following quantitative criteria:

Speed of development How long does it take to develop the analysis? Although this measure is not very exact, as different workloads and different developers blur the results, a tendency should be visible.

Code size How many lines of code does the analysis comprise?

Runtime of the analysis This is measured by comparing runtime results for identical setups, using the same program as input, the same analysis configuration and the same hardware for the respective approaches.

Comparing the results will offer insights on advantages and disadvantages of the approaches to incrementalization.

2.5 Chapter Summary

This chapter presented two approaches for the development of incremental static analyses: Manual incrementalization and automatic incrementalization.

The need for incremental static analysis as part of IDEs was discussed. Means to support development of manually incrementalized static analyses were presented by introducing Magellan as open platform for static analyses. Magellan allows to define analyses independent from each other, which is crucial in an open platform. Also, it enables to run static analyses along with the incremental build process offered by the Eclipse IDE. To enable the integration of independently developed analyses, a specification language for analyses was proposed to describe the dependencies among analyses. The dependencies between analyses are specified with respect to the logical view on the data that is processed by the analyses.

An environment that supports automatic incrementalization of static analyses was presented. An introduction to Datalog was given and XSB as logic programming environment was introduced, including incremental tabling, which enables automatic incrementalization of static analysis. The representation of Java code in Datalog was discussed. XSB is embedded into Magellan as a query engine.

Criteria for comparing the approaches were introduced. In the following chapters, static analyses will be presented and their implementations compared according to the presented criteria.

Chapter 3

Incremental Confined Types Analysis

This chapter shares some material with the paper Incremental Confined Types Analysis [54]

This chapter presents two approaches to incrementally check for violations of confined types [135]. The analysis is an example for an optional type system [22]. These type systems do not influence the runtime semantics, but flag certain errors at compile time. This allows multiple type systems to coexist and to be checked by static analyses, without interfering with the type system of the core language. Confined types complement the builtin Java type system and are used to enforce security properties. Having an incremental implementation of an analysis that checks for violations of confined types improves the speed of the analysis and enables its use as part of the incremental build process.

This chapter is structured as follows: The following section gives a motivation for the use of confined types and presents the contributions of this chapter to the state of the art. Section 3.2 discusses confined types. Section 3.3 presents the implementation of the automatic incrementalized confined types analysis. Section 3.4 describes the manually incrementalized implementation. Section 3.5 compares the approaches. Section 3.6 discusses related work and Section 3.7 summarizes.

3.1 Introduction

Unintended aliasing of objects causes many kinds of problems. For example, aliasing makes modular reasoning more difficult, as it is hard to reason about the effect of updating an object `o` when it is unknown which other objects

```
1 public class Class {
2   private Identity[] signers;
3   public Identity[] getSigners() {
4     return signers;
5 } }
```

Listing 3.1: Class.getSigners() without Confined Types

also keep a reference to o.

Besides hampering program comprehension, unintended aliasing can also lead to subtle errors. For example, an Enterprise Java Beans (EJB) container needs to have full control over the beans for the correct operation of its services, such as, pooling and persistence. An enterprise bean is not allowed to directly pass its **this**–pointer to other beans to avoid creating aliases that are not controlled by the container. E.g., the following situation will very likely cause an erroneous behaviour of the application: an enterprise bean passes its **this** reference to another object, then the container's instance pooling service (re)uses the bean to represent a different database entity and afterwards the bean—now representing a different entity—is directly accessed using the "old" **this** reference.

Besides being a source of programming errors that can be detected when testing an application, unintended aliasing can also lead to security errors, which are hard to detect using standard development techniques. For example, when a reference to an object is passed to another object and, hence, an alias is created for the first object, then the alias can later on be used to update the first object in an unanticipated manner. Vitek and Bokowski [135] discuss a security breach caused by a reference leaking bug in the JDK 1.1 (shown in Listing 3.1).

In the JDK's implementation, each instance of a Java Class object holds an array of signers (Line 2) that represents the principals under which the class acts. The problem is that the getSigners method returns a reference to the original signers array (Line 4). Hence, attackers can freely update the signatures based on their needs.

To solve the problems related to the creation of unintended aliases, means are needed to enforce that important data structures can not escape the scope of a well defined protection domain. For example, to assure that the reference to the original signers array does not escape the declaring class. To solve issues related to object aliasing, Vitek and Bokowski [135] propose the concept of confined types.

This chapter presents an incremental analysis for the confined types con-

cept and shows the integration of this analysis into the incremental build process of the Eclipse IDE. The original approach argued for language extensions to annotate the types. As one goal of the work is to ensure compatibility with the Java language specification and existing tools, Java annotations are used to achieve the necessary semantic extensions proposed by Vitek and Bokowski.

The main contribution is an implementation of the confined type checking that is tightly integrated with a standard software development environment and where the analysis exhibits a behavior that is indistinguishable from other (standard) compile time analyses. This fits well in the development philosophy supported by modern IDEs such as Eclipse, where the developer expects to see typing problems as soon as they emerge as the project evolves.

Thus, the confinement rules are implemented using the open, extensible static analysis platform Magellan [51], which is tightly integrated into the Eclipse IDE [47].

Checking program properties by IDEs avoids bloated compilers and ensures that application-specific checkers can be introduced when needed. However, (re)checking the entire project after a change is prohibitively expensive with respect to the time required for the analysis. Hence, violations of the typing rules for confined types should be checked for incrementally.

3.2 Confined Types

Confined types were proposed by Vitek and Bokowski [135] as a machine checkable programming discipline that prevents leaks of sensitive object references. A motivation for their work was the security breach mentioned in the introduction.

A possible solution to avoid the breach is a programming style that encourages the developers of classes with sensitive information to return a reference to a copy of the sensitive data, in this case a copy of the signers array. While programming styles cannot be enforced, using confined types ensures that none of the key data structures used in code signing escape the scope of their defining package.

For this purpose, types whose instances should not leave their defining package are marked as *confined*. *Confinement* ensures that objects of a confined type can only be accessed within a certain *protection domain*. A type is said to be confined to this domain if all references to objects of that type originate from within the domain. Code outside the protection domain is never allowed to manipulate confined objects directly. In contrast to existing access control mechanisms in Java (such as the Java **private** keyword), confinement

```
1  package java.security;
2    abstract class AbstractIdentity { @anon equals(){...};  }
3    @confined  class SecureIdentity extends AbstractIdentity { ... }
4    public class Identity {
5      SecureIdentity target;
6      Identity(SecureIdentity t) { target = t; }
7      ... // public operations on identities;
8    }
9    public class Class {
10     private SecureIdentity[] signers;
11     public Identity[] getSigners( ) {
12       Identity[] pub = new Identity[signers.length];
13       for (int i = 0; i < signers.length; i++)
14         pub[i] = new Identity(signers[i]);
15       return pub;
16     }
17   }
```

Listing 3.2: Class.getSigners() using Confined Types

constrains access to object references rather than classes. It prevents class-based restrictions from being circumvented by casting the protected object to one of its unrestricted supertypes.

Java packages are used as protection domains (as proposed in [135]). Instead of the new modifiers, confined and anon, introduced in [135], we use the metadata facility (*annotations*) introduced in Java 5.0 and define two annotation types: @confined and @anon.

Listing 3.2 shows, how the code from Listing 3.1 on page 54 can be rewritten using confined types. Classes whose objects should be confined to the containing package are tagged as @confined. In Listing 3.2, annotating SecureIdentity as @confined (Line 3) enforces references to SecureIdentity objects to be confined to the package java.security. Thus, code outside this package can never access instances of type SecureIdentity. Renaming the old Identity class to SecureIdentity and introducing a new Identity class (Lines 4 – 8) preserves the functionality of the original interface.

The @anon annotation enables confined types to safely use methods from unconfined types. Methods that do not reveal the current object's identity are marked as *anonymous* by annotating them with @anon to show this intention and to make this property checkable[1]. In Listing 3.2, the method equals

[1]Another possibility would be to infer the @anon property. But having it explicit as an annotation in the code serves as a documented design decision.

in Line 2 is marked with @anon to show that it never reveals the current instance's identity (**this**-reference). Therefore, SecureIdentity can safely extend AbstractIdentity and call equals on **this**, because no method marked @anon will breach the confinement.

The constraints in Table 3.1 and 3.2 are defined in [135] and lay down the semantics of `confined` and `anon`. Constraints in Table 3.1 restrict class and interface declarations (*C1*, *C2*), prevent widening (*C3*), hidden widening (*C4*, *C5*), and transfers from inside (*C6*) and outside (*C7*, *C8*) the protection domain. The rules defined in Table 3.2 constrain the usage of the self-reference **this** in method implementations, so that **this** is not revealed to code outside the method.

C1	A confined class or interface must not be declared public and must not belong to the unnamed global package.
C2	Subtypes of a confined type must be confined as well.
C3	Widening of references from a confined type to an unconfined type is forbidden in assignments, method call arguments, return statements, and explicit casts.
C4	Methods invoked on a confined object must either be non-native methods defined in a confined class or be anonymous methods.
C5	Constructors called from the constructor of a confined class must either be defined by a confined class or be anonymous constructors.
C6	Subtypes of java.lang.Throwable and java.lang.Thread may not be confined.
C7	The declared type of public and protected fields in unconfined types may not be confined.
C8	The return type of public and protected methods in unconfined types may not be confined.

Table 3.1: Constraints for confined types

A1	The reference **this** can only be used for accessing fields and calling anonymous methods of the current instance or for object reference comparisons.
A2	Anonymity of methods and constructors must be preserved in subtypes.
A3	Constructors called from an anonymous constructor must be anonymous.
A4	Native methods may not be declared anonymous.

Table 3.2: Constraints for anonymous methods

The best practise of returning only copies of sensitive data is supported by using confined types as an extension to the Java type system. Once a type is marked as @confined, the safety of the program with respect to avoiding unintended reference leaking can be guaranteed.

```
1 isConfined(Class):- classAnno(Class,annotation(ref('de.tud.magellan.
      confinedtypes.annotations.confined'),[])).
2
3 isAnon(MId):- methodAnno(MId,annotation(ref('de.tud.magellan.confinedtypes.
      annotations.anon'),[])).
```

Listing 3.3: CommonConfinedTypes-rules

3.3 Automatic Incrementalization

The implementation described in this section is based on the diploma thesis of Mathias Kahl [84].

This section describes the automatic incrementalization of the analysis using the Datalog engine described in Section 2.3. To represent the necessary facts about the program the source representation described in Section 2.3.2 is used.

The analysis requires the following facts, to represent all entities and relations that are mentioned in the Tables 3.1 and 3.2 on the preceding page.

- **source elements**: class, interface, field, method and their respective annotations.
- **statements**: method calls and returns; field and variable assignments and accesses, as well as casts.
- **relations**: inheritance relationships.

The total number of facts stored in the fact base was reduced by one third by configuring the DatabaseAnalysis to exclude the facts not needed for this analysis.

After transfering the source representation into the Datalog engine, a query for each of the confinement and anonymous rules is issued, and the results are presented to the developer as errors together with the errors reported by the Java compiler.

Listing 3.3 shows, how the annotations are read. isConfined(Class) checks, whether a given class is annotated as being confined. isAnon(MId) checks, whether a given method with identifier "MId" is annotated as being anonymous.

Listing 3.4 on the facing page shows two examples for the Datalog representation of the confinement rules. confined1(Class) holds if C1 is violated,

```
1  confined1(Class):-
2    isConfined(Class),
3    (public(Class)|
4    protected(Class)|
5    package(Class,'null')).
6
7  confined8(Class,Mname,Param,Returntype):-
8    isConfined(Returntype),
9    method(MId,Class,Mname,Param,Returntype),
10         tnot(isConfined(Class),
11         (public(Mid)|protected(Mid)).
```

Listing 3.4: Confined types-queries

i.e., if a class is confined and is either public, protected or in the default package. confined8(Class,Mname,Param,Returntype) holds, if C8 is violated, i.e., if a type that is marked as being confined is returned by a public or protected method in an unconfined type.

```
1  anon2(Class,Mname,Param,Return) :-
2    isAnon(MId),
3    method(MId,Cl,Mname,Param,Return),
4    Mname\=='<init>',
5    method(Subclassmnr,Class,Mname,Param,Return),
6    inherits(Class,Cl),
7    \+isAnon(Subclassmnr),
8    \+isConfined(Class).
9
10 anon4(Class,Mname,Param,Return) :-
11   isAnon(MId),
12   method(MId,Class,Mname,Param,Return),
13         native(Mid),
14   \+isConfined(Class).
```

Listing 3.5: Anonymous -queries

Listing 3.5 shows examples for the Datalog representation of the anonymous rules. anon2(Class,Mname,Param,Return) holds, if an anonymous method is overwritten by a non-anonymous method. anon4(Class,Mname,Param,Return) holds, if a native method is declared anonymous.

The other rules are checked similarly. For incremental builds, the facts for deleted source code elements are revoked and the facts for added source code

```
1  package x;
2    public class X1 {
3      @anon public void m() { /* ... */ }
4    }
5    public class X2 {
6      public void m() { /* ... */ }
7    }
8
9  package y;
10   public class Y extends X1 { } /* change: ... extends X2 */
11
12 package z;
13   @confined class Z extends Y { /* ... */ }
14   class W {
15     public void foo() {
16       Z z = new Z();
17       z.m(); /* will violate C4 after change */
18   } }
```

Listing 3.6: Indirect violation of confinement constraints

elements are added to the Datalog data base. Then, the incremental table maintenance is triggered, and the queries for the confinement and anonymous rules are repeated and the list of errors is updated.

3.4 Manual Incrementalization

This section describes the manual incrementalization of the analysis using Java analyses integrated into Magellan.

Checking the confinement rules is modular in the sense that each class can be analyzed separately [135]. However, determining which classes have to be reanalyzed after a set of arbitrary changes to the project's source code is non-trivial. For an example of how a small change can impact the confinement rules at a seemingly unrelated location consider Listing 3.6.

The example consists of Java classes in three different packages. Class W calls a method m on a confined class Z. $C4$ is satisfied because Z inherits m from class X1 where it is declared anonymous. Now, let us assume that Y is changed to inherit from X2 instead of X1. Since X2 does not declare m as anonymous, the method call in Line 17 now violates constraint $C4$. Hence, a change in package y (which does not contain any confined or anonymous declarations) yields a confinement error in a class in package z that is neither

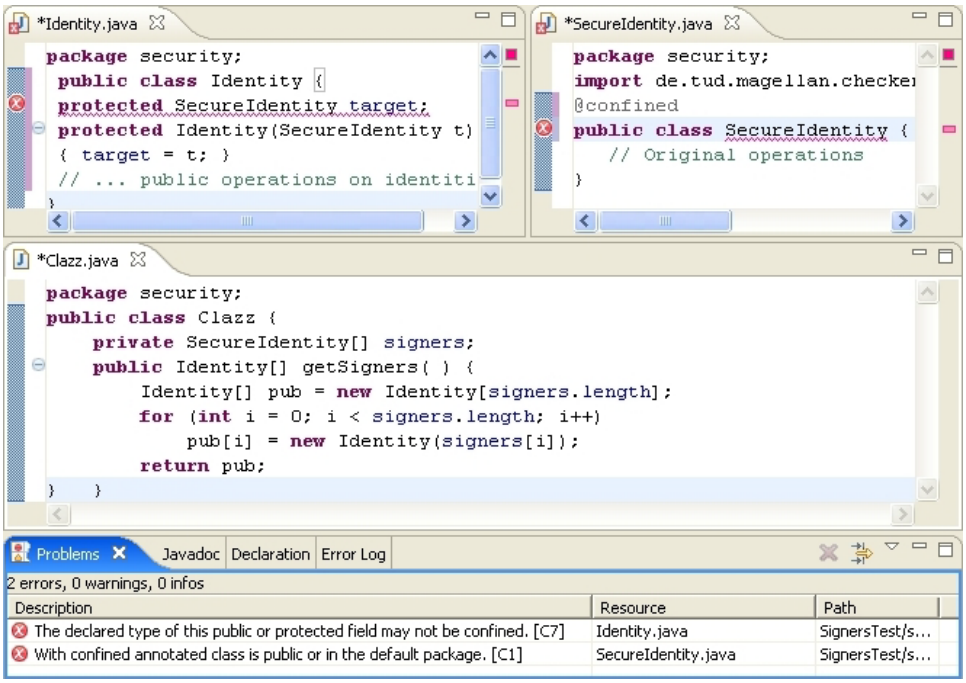


Figure 3.1: Screenshot of Eclipse when using confined types

a subtype nor a supertype of the changed class Y.

The example shows that when a class changes, it is not sufficient to only check classes in the same package and therefore the same protection domain or all supertypes and subtypes of the changed class. Therefore a more systematic approach is used to develop an incremental algorithm for checking the confinement rules.

The checking algorithm works in the following two steps:

1. Given a list of classes that have been changed, the extent of the change is identified, i.e. a set of classes that must be reanalyzed to discover any new constraint violation and to remove any error message for constraints that are no longer violated.

2. The constraint rules are checked for all classes in the extent that are returned by the first step. Whenever a check fails, an error report for the Eclipse problems view is created and presented to the user (see Fig. 3.1). Hence, after editing a source file the developer is immediately informed about constraint violations.

The constraints from Tables 3.1 and 3.2 on page 57 can be seen as predicates over classes and methods. For any class x,

- $C_i(x)$ is true, if and only if x satisfies C_i for any method m.
- $A_i(m)$ is true only if m satisfies the constraint A_i.

Each predicate can be evaluated on its own, since the definitions of the constraints do not depend on each other. For example, for a class x to satisfy constraint $C4$ it suffices that methods called on confined types within x are declared as anonymous. Whether these methods, in turn, satisfy the constraints for anonymous methods is irrelevant for $C4$, because error messages are directly related to the violated predicates. Violations of the constraints for anonymous methods will be displayed as separate errors when analyzing the respective methods.

The problem can now be reworded as follows: Given a program, the predicate values for all its classes and methods, and a set of classes changed in the process of an incremental build, update the predicate values so that they reflect the program changes. This update process should be *correct* in the sense that it produces the same results as a whole-program analysis.

Since a constraint only needs to be reevaluated if some information it depends on has been invalidated by a program change, we determine for each constraint the set of information it depends on.

Before doing so, the following constraints need to be slightly modified:

- $C2$ is changed to $C2'$: "If a direct supertype of a type t is confined, t must be confined as well.",
- $A2$ is changed to $A2'$: "If a method m directly overrides an anonymous method, m must be anonymous as well."

These modifications, while reducing the information on which the values of $C2$ and $A2$ predicates depend on, do not affect the semantics of the confined types: A program satisfies all the constraints from Tables 3.1 and 3.2 on page 57 if and only if it satisfies them with $C2$ and $A2$ replaced by $C2'$ and $A2'$.

The analysis is started by investigating the rules for anonymous methods, as defined in Table 3.2.

- $A1(m)$ depends on the anonymous attribute of all methods called on `this` inside m. These methods have been declared either in m's class or in a supertype of the latter. Hence, for any changed class x, $A1(m)$ must be reevaluated for any m in x or any of its subtypes.
- $A2'(m)$ depends on the anonymous attribute of the method overridden by m. Since such a method must be declared in a supertype of m's class, the same invalidation strategy as for $A1$ applies.

- Since calls to constructors from within a constructor can be seen as a special kind of method calls on `this`, $A3$ is treated in the same way as $A1$.

- $A4$ does not depend on any non-local information. Thus, it suffices to reevaluate $A4$ on all methods of a changed class.

This leads to the following incremental algorithm for checking the constraints from Table 3.2 on page 57. Whenever a type t changes, constraints $A1$–$A3$ on all subtypes of t (including t itself) are reevaluated. Constraint $A4$ only has to be reevaluated for types that have been changed.

Next, the constraints in Table 3.1 on page 57 are analyzed in the same way.

- $C1(x)$ only depends on information from the class x. Thus, for every x, which has changed, $C1(x)$ must be reevaluated.

- $C2'(x)$ depends on the confined attribute of all direct supertypes of x. Thus, $C2'(x)$ is reevaluated for any class x that is a direct subtype of a changed class (but not for the class itself).

- $C3(x)$ depends on the confined attribute of the types used in widenings inside one of x's methods. The value of $C3(x)$ can change only if either x is changed (so that the list of widenings performed inside x has changed) or if the confined attribute of a type t that is used in a widening changes. For each such t, the following holds: t has been confined at some point (i. e., before or after the change), hence, t is defined within the same package as x. Therefore, for each class x whose confined attribute has changed $C3$ needs to be reevaluated for any class in the same package as x.

- $C4(x)$ depends on method calls within x where the static type of the receiver is confined. More specifically, it depends on the confined attribute of the method's declaring type and the method's anonymous attribute.

  Since the static receiver type is confined, it must be in the same package as the class that contains the method call. Thus, whenever the confined attribute of a type t changes, $C4(x)$ must be reevaluated for any class x in the same package as t to recheck all relevant method calls on t.

  Additionally, $C4(x)$ is reevaluated when the anonymous attribute of the called method changes. This can happen indirectly as seen in the example from Listing 3.6 on page 60. Thus, whenever a type t is

changed all classes are determined that call a method on a confined subtype t' of t. Since a confined type can only be package visible, such a class must be in the same package as t'. For every confined subclass t' of t we check $C4(x)$ for all classes x in t''s package.

- The constraint $C5(x)$ considers constructor calls in constructors of confined classes. Since constructors are not inherited in Java, they have to be in the same class or in the direct superclass (can be called via **super**(...)). This implies that $C5(x)$ depends only on x itself and its superclass. When a class x is changed, $C5(x)$ is reevaluated for x and all direct subtypes.

- $C6(x)$ depends on all superclasses of x. Thus, whenever class x changes it suffices to reevaluate $C6(x)$ for all subclasses of x. As an optimization, changes to x that do not change x's supertypes can be ignored.

- $C7(x)$ can change whenever the confined attribute of a type used in a public or protected field declaration of x changes. Since such a field type either was confined before the change or has become confined after the change, it has to be in the same package as x. Thus, whenever a type t changes $C7(x)$ needs to be reevaluated for all classes in the same package as t.

- The constraint $C8(x)$ checks return types of methods that are declared as public or protected. The strategy for evaluating $C8(x)$ is the same as for $C7(x)$.

Given a set of files that have been changed, every constraint is processed separately. For every changed class, the set of classes is computed that have to be reanalyzed and then the constraint is reevaluated against all classes in this set. For simplicity, the union of all these sets is computed and all constraints are checked against every class in this set. This process is correct even if multiple changes have been performed, because it analyzes the same classes that would have been analyzed if an incremental analysis had been performed after every change.

By definition, the rules for computing the set of classes to be checked after a change guarantee that a constraint is reevaluated if any information it depends on has been invalidated. Hence, the value of all predicates is the same as if they had been evaluated by performing a whole-program analysis. Thus, the incremental algorithm is correct.

3.5 Comparison of the Approaches

To compare the implementations using the approaches, their respective runtimes was measured while editing a software project inside Eclipse. First, the setup of the experiment is discussed. The setup is identical for both implementations. Then the measurements for the two implementations are detailed. Section 3.5.2 presents the measurements for the automatically incrementalized implementation and Section 3.5.3 presents the measurements for the manually incrementalized implementation. In Section 3.5.4 conclusion are drawn from the experiment.

3.5.1 Setup

The experiment was conducted on an Athlon 2.6 Ghz workstation with 1 GB RAM and Sun Java 5 JDK. The project used is the BAT Bytecode toolkit [50]. At the time of the experiment, BAT comprised 22 packages, 790 classes, 45 interfaces, and 7,750 methods.

The test set was supplemented by 17 classes from a second project spread over 3 packages which implement a small part of a public key infrastructure. The second project was used as small test bed during development. Initially, confined types were used in two of the packages. When performing the changes, classes in the third package were also made confined. Initially, 26 errors related to confined types were present in the code.

The source code changes are designed to simulate usual edit actions during development. The changes were performed using both implementations and are described in detail below. Table 3.3 on the next page shows the size of the changes: The first column numbers the builds executed after the changes described below. The next two columns show, how many classes and methods are added and removed in the incremental build. The fourth column shows the number of facts that were added and removed to update the Datalog database while running the automatically incrementalized implementation. The last column shows the number of violations that remain after the edit.

The following code changes were performed for the corresponding build number:

1. Generated a public getter-method for a confined field resulting in one new $C8$ violation.

2. Declared a class as confined, resulting in one new $C1$, one new $C4$ and one new $C5$ violation; one $A2$ and one $A4$ violation disappeared.

Build	added / removed classes	added / removed methods	added / removed facts	violations
1	1 / 1	2 / 3	13 / 16	26
2	1 / 1	4 / 4	22 / 23	27
3	1 / 1	4 / 5	92 / 95	27
4	1 / 1	2 / 2	7 / 8	29
5	1 / 1	9 / 10	31 / 33	30
6	1 / 1	15 / 15	109 / 110	47
7	1 / 1	20 / 20	189 / 189	47
8	3 / 3	14 / 14	130 / 131	48
9	2 / 2	11 / 22	70 / 93	44
10	8 / 8	115 / 115	1,586 / 1,586	44
11	0 / 1	0 / 3	0 / 10	44
12	2 / 2	18 / 18	120 / 120	32
13	2 / 0	4 / 0	20 / 0	31
14	1 / 1	7 / 7	51 / 49	28

Table 3.3: Properties of Code Changes

3. Applied "Extract method..." refactoring. No violation changes occured.

4. Declared a public class as confined, resulting in one new $C1$ and one new $C6$ violation. A $C2$ violation was replaced by a different one.

5. Added a **native** anonymous method, resulting in an $A4$ violation.

6. Declare a class as confined, resulting in 17 new violations: one $C1$ and four $C4$ violations. Additionally, a $C2$ violation appeared for each of the 12 subclasses of the confined class.

7. Changed a comment. No violation changes occured.

8. An anonymous method was changed to invoke a non-anonymous method, resulting in a new $A1$ violation.

9. A confined class was changed to no longer extend java.lang.Throwable and to implement a new interface, resulting in eleven new methods. This removes three $C6$ and one $C4$ violation.

10. Renamed a class. No violation changes occured.

Build	untabled XSB [ms]	tabled XSB [ms]	incrementally tabled XSB [ms]
1	1,229	666	176
2	1,290	745	195
3	1,386	664	171
4	1,408	699	168
5	1,531	702	194
6	6,017	1,117	555
7	5,883	1,122	210
8	6,030	1,235	348
9	5,853	1,245	291
10	6,210	1,408	663
11	5,831	1,078	144
12	5,901	1,206	297
13	5,745	1,120	216
14	5,611	1,049	230
average	4,389	1,036	294

Table 3.4: Effects of Incremental Tabling

11. Added a new abstract class with two constructors from its superclass. No violation changes occured.

12. A confinement annotation is removed from a class and added to an inheriting class resulting in one new $C2$ violation and twelve $C2$ violations in subclasses.

13. Deleted two unused classes. One $C4$ violation disappeared.

14. Corrected all three errors ($C1$, $C5$, $C7$) of a class by removing a confinement annotation and making four methods anonymous.

3.5.2 Automatic Incrementalization

In case of a full build, creating the Datalog facts takes 3,300 msecs. This comprises the transformation of the Java files into the 3-address representation and the creation of the Datalog facts; to add the generated facts to the database, XSB requires another 5,200 msecs. Since all tables are initially empty, the first evaluation of the queries takes 328 msecs.

Adding the 45.7 seconds for the supporting analyses, adds up to 54.5 seconds for the full build.

Memory consumption has not been observed in detail because the current version of the incrementalized XSB does not free unused memory. It allocates a certain amount of memory at the beginning and re-allocates memory on demand. For the analyzed test data about 100 MB were occupied.

Table 3.4 on the preceding page compares the runtimes for the automatic incrementalization (in the rightmost column) with untabled XSB (in the second column) and tabled XSB (in the third column). In comparison to non-incremental evaluation, the system is between 1.4 and 8 times faster. In case of non-incremental evaluation, the queries need to be reevaluated from scratch after every change; in particular it is necessary to explicitly delete all tables, as the tables are not maintained incrementally. Most of the time required by the incremental build goes to maintain the tables. This time is largely dependent on the number of facts that need to be removed and added and thus scales with the size of the changed code.

Table 3.5 on the next page shows the performance of the Datalog based approach. The first column shows the executed build. The second column shows the time needed by both approaches to transform Java byte code into the 3-address representation in SSA form [7]. The time to create the Datalog representation is presented in the third column. Column four shows the time used by the incrementalized XSB engine. The last column shows the results for the manually incrementalized implementation and is discussed in the next sections.

The automatically incrementalized implementation executes the builds in generally well below one second and thus is fast enough to execute along with the incremental build process for projects with at least 1,000 classes. Even in case of changes that affect large numbers of facts (builds 6, 7, 8, 10 and 12) the execution times are acceptable.

3.5.3 Manual Incrementalization

The overall time for the full build of the project is 46.5 seconds; the supporting analyses require 45.7 seconds and the analysis of the confined types (`Confinement Analysis`) 0.7 seconds.

In Table 3.5 on the facing page the runtimes of the Java based manually incrementalized analysis are shown. The second column shows the time needed to prepare the code representation and the last column shows the runtime of the confined types analysis. The results show that in case of an incremental build the time required to perform the necessary analyses is in general less than 200 milliseconds. The automatic parallelization of the artifact processors reduces the required time for processing the source files by $\approx 35\% - 40\%$ when compared to a single CPU configuration. Further,

Build	create code representation (for Java and Datalog) [ms]	create Datalog encoding [ms]	Datalog based analysis [ms]	Java based analysis [ms]
1	7	0	176	3
2	8	0	195	5
3	36	1	171	2
4	6	0	168	6
5	8	0	194	2
6	12	0	555	124
7	36	1	210	3
8	14	1	348	4
9	11	1	291	4
10	211	20	663	81
11	121	0	144	1
12	11	1	297	84
13	3	0	216	7
14	16	0	230	3

Table 3.5: Comparison of the measurements for both implementations

the additional amount of memory required is at most 85 MB. These results indicate that it is feasible to run the manually incrementalized confinement analysis along with the incremental build process.

3.5.4 Conclusions

As the performance figures show, the overhead when always executing the analysis along with the incremental build process is in both cases low enough to be able to use confined types in day-to-day usage.

The Datalog based approach took comparatively little effort (two days) to implement the queries. The Java based approach has hand tuned checkers which are harder to develop. The Java checkers are faster as they are optimized manually, but their development took about 10 times longer.

The Datalog queries comprise about 280 lines of code. The Java approach comprises about 2,500 lines of code.

The Java based analysis is between four and 185 times faster. The following properties favor the Java based analyses:

- The analysis is modular on class level, so that each class can be analyzed separately. Apart from the inheritance hierarchy, no whole-program

facts need to be maintained for this analysis.

- The extent of source code changes is relative small and easy to compute.
- The configuration for confined types is done by marking types as confined and methods as anonymous and thus explicitly enumerating the source elements for the analysis.

3.6 Related Work

When dealing with aliasing, four categories of work are considered [79]: detection, prevention, control and advertisement of aliasing. The works relevant for the analysis discussed in this chapter mostly fall under the category of prevention and control.

The notion of alias protection for object-oriented languages was introduced by Hogg [78] in order to enable modular reasoning for groups of classes. These groups are called *islands* and ensure the restriction of aliasing to classes on the island. Hogg differentiates between static and dynamic aliases. Static aliases are aliases via instance variables and dynamic aliases are those via parameters or local variables. Static aliasing can lead to undesired side effects in later invocations of the aliased object. Dynamic aliases were seen as unproblematic, because they disappear at the end of the execution of the method in which they are defined. Means to control static aliasing were introduced with islands. Islands are the transitive closure of a set of objects accessible from a *bridge* object. A bridge object is the sole access point to a set of instances that make up an island.

To ensure that no static aliases are created from outside the island to objects on the island, the methods of the bridge object are restricted. Only methods with parameters and return values that either do not modify the state of the system, or have only parameters and return values that have at most one static alias are allowed. This avoids the creation of unwanted aliases. For example, a return value of a method can be tagged with *unique* to state that exactly one reference to its value exists. The value can only be assigned to other variables, if the original reference is released.

The full encapsulation of aliases of this approach is too restrictive for many common design idioms used in OO programming. E. g., no object could be a member of two collections simultaneously if either collection was fully protected against aliases. In this case, one collection would be an island, prohibiting that references to its members show up outside the island.

Noble et al. [109] present a more flexible approach to control aliasing when compared with *islands*. Their approach controls aliasing by introducing ex-

plicit aliasing modes. The authors differentiate between the *representation* of an object, which corresponds to its fields, and *arguments*, which are parameters to methods of the object. The representation of objects should only be accessible via the object's interface, e. g., in Java fields would have to be marked as `private` and aliases to them should not be returned via getter methods. The state of the object should only depend on arguments with an immutable state. If the state of the object was dependent on the mutable part of arguments to its methods, the state of the object could be modified by changing the state of the arguments long after the call, bypassing the objects interface. The approach uses tags to annotate types and enables the compiler to enforce the restrictions mentioned on the creation of aliases. A formalization of this model is discussed by Clarke et al. [36]. Even though both approaches enable flexible alias control, they are designed for a language without inheritance or subtyping.

A variant of ownership types is used by Boyapati et al. [21] to prevent data races and deadlocks by partitioning locks into a fixed number of equivalence classes and specifying a partial order among these equivalence classes. The type checker then statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in descending order. Ownership types are used to ensure that that the locks that protect an object also protect its encapsulated objects.

Clarke et al. [35] implement a confinement checker for Java to solve the domain specific problem of passing a **this** reference from one Enterprise Java Bean component to another component. In EJB access to the internal objects implementing each bean must be prevented, and access to the bean is permitted only through the container generated wrapper. While confined types are a generic solution to control aliasing, Clarke et al.'s approach solves an EJB specific problem.

Fong [65] describes how to translate the notion of confinement, which is formulated for static analysis of Java source code, to dynamic analysis of Java Bytecode. The approach retains the confinement annotations made in the source code at bytecode level. This enables link time checks of confinement rules. It also describes a form of secure cooperation between mutually suspicious code units, where, for example, a resource object can be shared between two untrusting modules while ensuring its confinement to a given domain. The implementation extends the runtime of the Pluggable Verification Modules of the Aegis Research JVM. The approach discussed in this chapter uses static analysis to ensure the confinement properties at compile time and to immediately inform the user of confinement violations.

The notion of confined types is formalized by Zhao et al [138] in the context of Featherweight Java. In Featherweight Java, confined types are

extended to confined instantiations of generic classes.

Reverse engineering approaches to the detection of aliasing are described by Grothoff [68] and Potanin [112]. Grothoff developed Kacheck/J [68] as a tool to infer confinement in Java code. Kacheck/J was used to test the thesis that all package-scoped classes in Java programs should be confined. About 25% of the classes of their benchmark suite respected the confinement rules anyway and 45% could be refactored to be confined just by changing visibility modifiers. These numbers are supported by the findings of Potanin et al. [112]. They presented metrics of uniqueness, ownership and confinement by analysing snapshots of Java program's object graphs and found that a third of all objects were strongly confined.

3.7 Chapter Summary

In this chapter, incremental confinement analysis was discussed. The need for pluggable type systems to check domain specific properties was introduced. The confined types analyses was introduced as a machine checkable programming discipline that prevents leaks of sensitive object references.

Two approaches for the implementation were discussed:

- A Datalog based approach using XSB as engine, that supports automatic incrementalization.
- A Java based approach, using manual incrementalization.

The comparison of the two approaches showed, that the analysis effort to develop the Java based analysis surpasses the Datalog based approach by an order of magnitude, measured in lines of code and development time. Considering the runtime, the Java based approach is roughly 50 times faster than the Datalog based approach. This is due to the modular nature of the analysis, requiring only the type hierarchy as hole program fact. The configuration is based on tags annotating methods and types and therefore changes to the configuration are evaluated easily.

Chapter 4

Incremental Call Graph Analysis

The whole program call graph is a key data structure in static analysis. Besides using the call graph directly for, e.g., visualizing the control flow between methods, the call graph can be used as input for other analyses. For example data–flow analyses [3], points–to analyses [91], or escape analyses [33] all need call graphs as part of their input. Incrementally maintaining the call graph instead of recomputing it from scratch speeds up the analysis time for the call graph significantly. It also provides change sets to the call graph, which enables the development of incremental algorithms that build on the call graph.

This chapter describes an approach to incrementally maintain the whole-program call graph. The next section gives an overview about the area of call graph analyses and sets the context for the chosen algorithm. Section 4.2 gives a short overview about algorithms for call graph construction and describes the call-graph algorithm chosen to be incrementalized. Section 4.3 shows an approach to automatic incrementalization for the algorithm. Section 4.4 describes the manually crafted incrementalization of the algorithm. Section 4.5 compares the approaches. Section 4.6 discusses related work and Section 4.7 summarizes the chapter.

4.1 Call Graphs

The call graph is a directed graph, that represents the calling relations among methods of the program. The nodes in the graph represent methods; the edges between the nodes represent calls between methods.

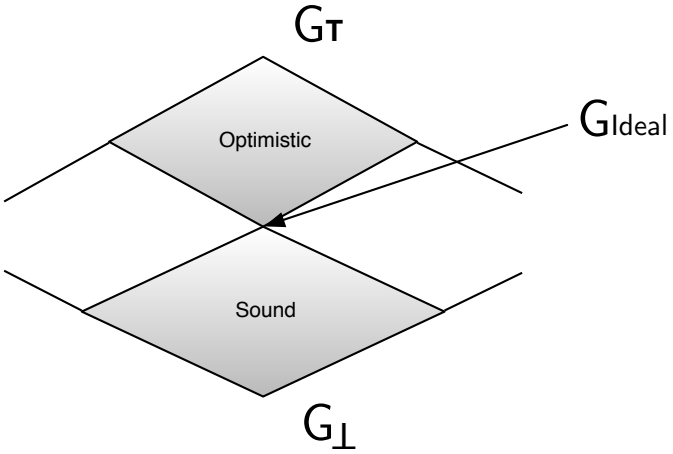


Figure 4.1: Regions in a Call Graph Domain

4.1.1 Comparing Call Graphs

Figure 4.1 (taken from [70]) shows a lattice that orders call graphs according to their precision. At the top, there is the call graph $G_\top$, in which no node is connected to any other node. At the bottom is the call graph $G_\bot$, which, in which each node is connected to each other node. The graph in the middle G_{ideal} is the ideal call graph, where each edge in the call graph corresponds to a call in at least one execution and for each call in an execution, there is an edge in the call graph. Call graphs that reflect a particular execution are called optimistic and are located above the ideal call graph. The call graphs below this ideal graph are called sound. For a call graph to be sound, it must safely approximate any program execution, hence G_{ideal} is the most optimistic sound call graph.

Unsound call graphs are less useful as basis for error detection analysis, because they increase the amount of false negatives. If, for example, an analysis that searches for potential null-pointer resolutions would use an unsound call graph, control flow that would lead to a null-pointer exception could go undetected. On the other hand, the call graph should be as close to the ideal call graph G_{ideal} as possible, because these call graphs are smaller and are faster to use.

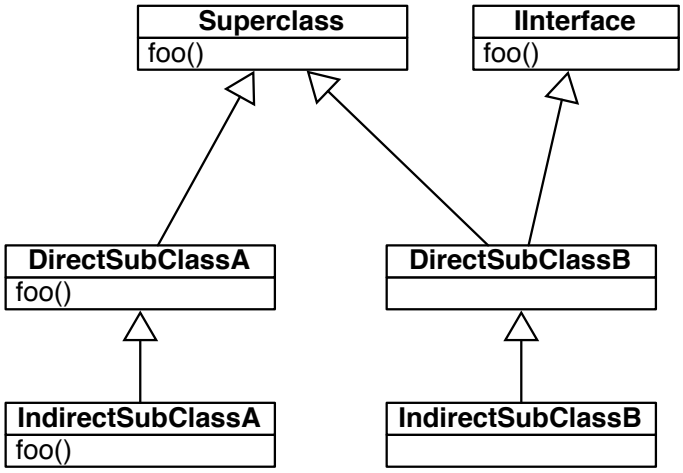


Figure 4.2: Sample Class Hierarchy to Show Virtual Call Resolution

4.1.2 Program Virtual Call Graph

The Program Virtual Call Graph (PVG) is a whole program call graph, that is used to hide language-dependent aspects from call graph construction algorithms. That includes Java byte code specifics, like method visibility or inheritance. To build the PVG, virtual method calls are resolved using class hierarchy information from the Class Hierarchy Graph (CHG), which provides the inheritance relation among types.[1]

The nodes of the PVG are *executable (concrete) methods*. To each method node, a set of *call sites* is attached. Call sites represent method calls and contain the source location of the call. For an example, see the class hierarchy in Figure 4.2 and the Listing 4.1 on page 77. Figure 4.3 on the following page shows the corresponding PVG. The method Test.test() contains three call sites to methods named foo(). Call sites can be direct or virtual. *Direct call sites* are call sites, whose method dispatch can be statically determined. For *virtual call sites*, the method dispatch has to be done dynamically. *Instantiating call sites* represent constructor calls.

The edges in the PVG are *call instances*. The call instances that represent the method calls are printed as arrows in Figure 4.3, labeled with their sets of receiver types. Each call instance has a set of possible receiver types. Therefore, each call site contains zero[2] or more call instances. Call instances

[1] In Java, a type T_1 *inherits* from a type T_2, if T_1 extends or implements T_2.

[2] for incomplete programs, e.g. calls to abstract methods with no implementation.

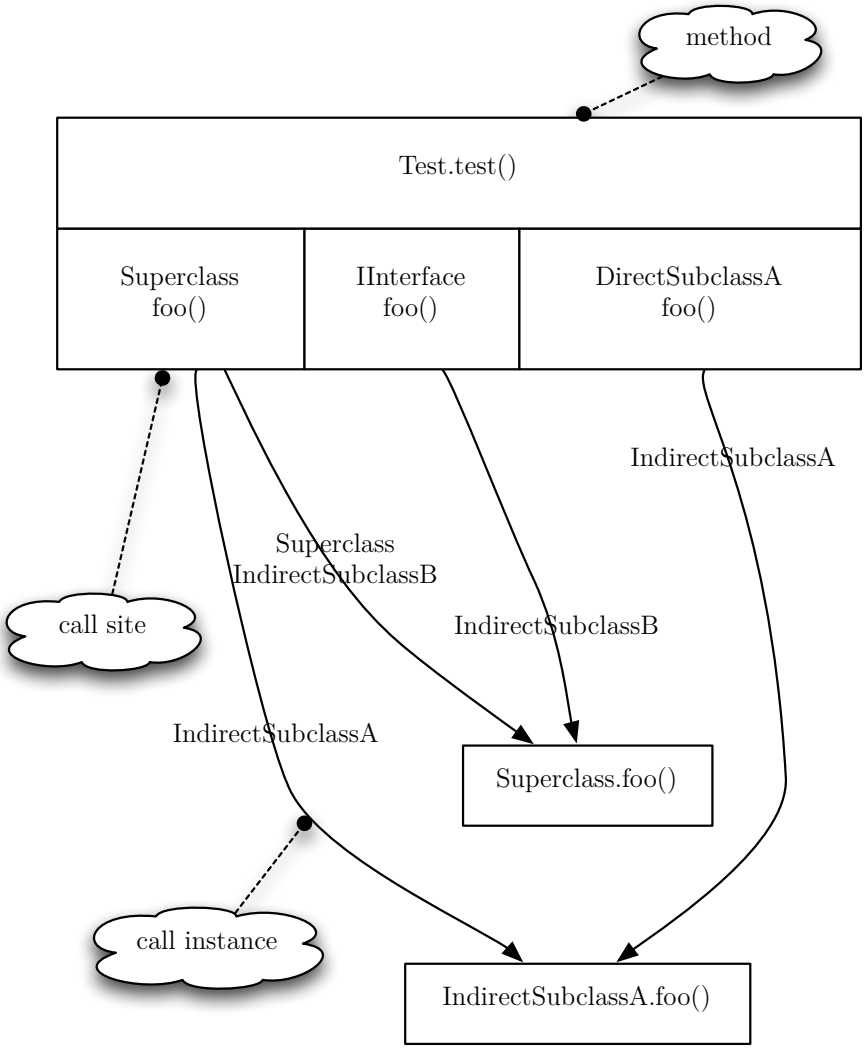


Figure 4.3: Program Virtual–Call Graph

```
1  class Test{
2    Superclass s = new Superclass();
3    IInterface i = new DirectSubclassB();
4    DirectSubclassA a = new DirectSubclassA();
5    void test(){
6      s.foo();
7      i.foo();
8      a.foo();
9    }
10   void a(){
11     f(s);
12     f(a);
13   }
14   void f(Superclass x){
15     x.foo();
16   }
17 }
```

Listing 4.1: Sample Program to Show Virtual Call Resolution

can be *direct* or *virtual*. Direct call instances are call instances of direct call sites, and have a single receiver type that is statically determined.

A virtual call instance represents a possible receiver of a dynamically dispatched method call. Such a virtual method call may point to many possible target methods. For example, the call site invoking Superclass.foo() in Test.test() leads to a call instance for each method "foo()" in subclasses of "Superclass" that overrides "Superclass.foo()".

The PVG thus is comprised of the following sets:

- concrete methods M
- concrete types T
- call sites S and
- call instances I.

4.2 Algorithms for Call Graph Construction

There are various algorithms for constructing a call graph. The algorithms differ in the precision of the resulting call graph and the time and space complexity of the algorithms [69].

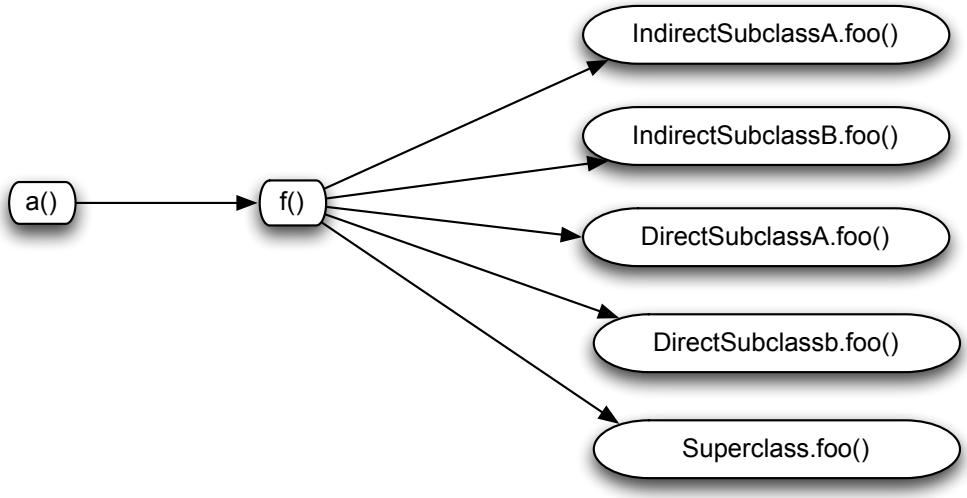


Figure 4.4: Call Graph according to a Context–Insensitive Algorithm (CHA)

4.2.1 Comparing Call Graph Construction Algorithms

Call graph algorithms can be categorized into two classes [108, p. 95]:

- Context–insensitive algorithms analyze each method once for all possible calling contexts (e.g., other method calls).
- Context–sensitive algorithms perform different analyses for different calls of the same method.The information obtained for a method always refers to the context of its analysis.

For an example, consider the two calls to Test.f() in method Test.a() in Listing 4.1 on the previous page. Context–insensitive algorithms would construct one node in the call graph for f(), as the calling context does not influence the call graph. An example is shown in Figure 4.4. The boxes represent methods and the arrows between the boxes represent calls. Context–sensitive algorithms construct one node for f() per calling context. In the example, two nodes are constructed, as visualized in in Figure 4.5 on the next page.

Context sensitive analyses lead to a better precision of the obtained call graphs, but have longer runtime than context–insensitive analyses. In the example, for the second call to f(), the more precise type in the parameter influences the call graph segment for the second call to f() and reduces the potential call targets from five nodes to one node. Most context–sensitive algorithms require a fix–point iteration, with a complexity of at least $O(n^2)$

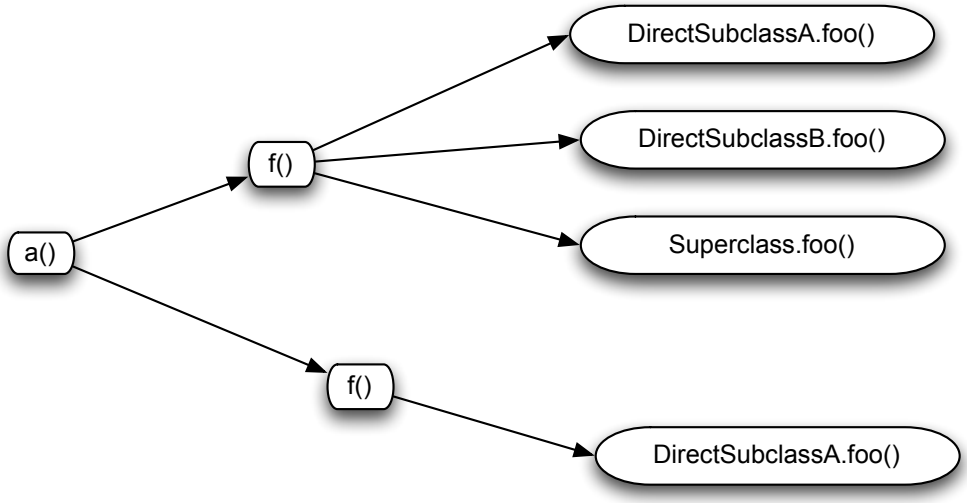


Figure 4.5: Call Graph according to a Context–Sensitive Algorithm (0–CFA)

with n as the number of methods [69]. As the focus of this work is on static analyses that can be integrated into the incremental build process of IDEs, context–insensitive analyses are examined, due to their better runtime complexity.

There are different kinds of context insensitive call graph construction algorithms [130]. The following kinds are relevant for this work and are presented in increasing order of precision:

RA The *Reachability Analysis* with name-based resolution maintains a conservative set of reachable methods for the whole program, where "conservative" means that all methods are initially assumed to be unreachable. Starting at the entry points of the program each reachable method is added to the set. Method calls are resolved only by their name and signature. As an example, consider the call to i.foo() in Line 7 in Listing 4.1 on page 77. As depicted in Figure 4.6 on the next page, the call is resolved to all types having a matching method foo().

CHA In *Class Hierarchy Analysis* the resolution process is extended by taking into account class hierarchy information [40]. The dynamic receiver types of virtual method calls are approximated by static receiver type and class hierarchy information. Again, consider the call to i.foo() in Line 7 in Listing 4.1 on page 77. As depicted in Figure 4.7 on the next page, the call is resolved to the method foo() in IInterface and all its sub types.

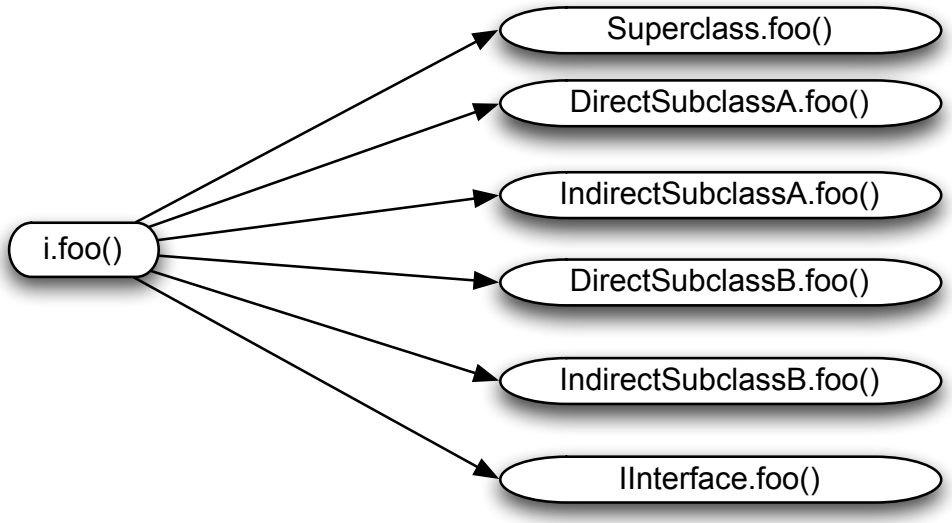


Figure 4.6: Call Graph fragment from RA Algorithm

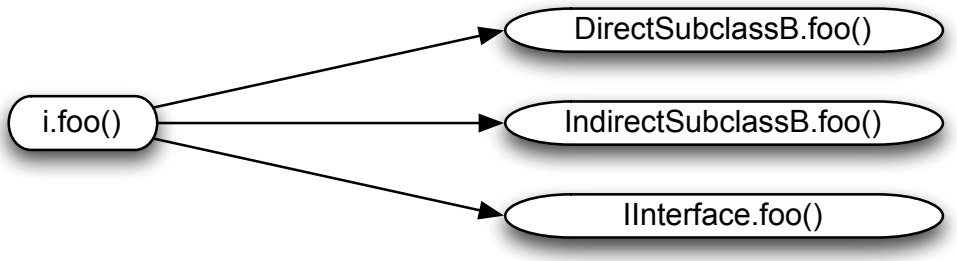


Figure 4.7: Call Graph fragment from CHA Algorithm

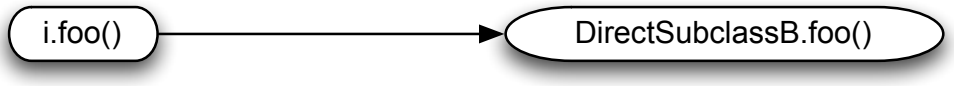


Figure 4.8: Call Graph fragment from RTA Algorithm

RTA The *Rapid Type Analysis* [13] resolves calls as the CHA and maintains a set of reachable methods, as RA does. RTA maintains a second, conservative set of instantiated types for the whole program, to exclude calls to methods whose classes are never instantiated. The term *live* is used for the reachability of methods and for the reachability of type instantiations. "Live functions are those that may be invoked during any execution of the program; live classes are those that may be instantiated during any execution of the program" [13, p. 3]. The call to i.foo() in Line 7 in Listing 4.1 on page 77 is again resolved to the method foo() in IInterface and all its sub types. But from these, only DirectsubclassB is instantiated and therefore, the call graph depicted in Figure 4.8 results. With the set of reachable type instantiations, the set of possible dynamic receiver types of a call is restricted to possibly instantiated types. If no possible dynamic receiver type of a call instance can be instantiated in any program execution, the call instance can be excluded from the call graph. Excluding those call instances increases the precision of the call graph without making it unsound.

0–CFA The *zeroth order control flow analysis* [124] maintains a set of targets for each expression in the program. Instantiated classes are propagate through flow graph starting with main and top-level **new** expressions. This increases the precision but increases the necessary amount of memory and runtime. 0–CFA has a complexity of $O(n^3)$, with n being the number of methods in the program [108]. As an example consider the calls to f() in Lines 11 and 12 in Listing 4.1 on page 77 as shown in Figure 4.5 on page 79. The flow of instantiated classes from their construction as attributes of the class to the calls of f() limit the call targets to the shown methods.

Most work on static analysis show the correctness of the analysis using simple language kernels. These language kernels lack features like implicit control flow due to callbacks from the virtual machine. Extending these algorithms to work with current, real programming languages—such as Java—requires substantial work and comprises some interesting questions, such as

how to deal with static initializers and finalizers.

Tip and Palsberg [130] compare precision and runtime of algorithms in the space between RTA and 0–CFA. Their results state, that using more than a single set to approximate the call targets of expressions increases the precision (calculated as number of edges) by 3%–20%, but increases the runtime about the factor of five. The RTA uses a global set, whereas 0-CFA uses a set per expression. Therefore, RTA is chosen as call graph analysis that represents a good compromise between speed and accuracy.

The rest of this chapter describes, how the Rapid Type Analysis [13] can be changed to work incrementally for Java. The original analysis was proposed for C++ and does not work incremental. The analysis introduced is extended to handle Java language specifics correctly and provides a general mechanism to account for the liveness of methods caused by implicit control flow.

To prove the concept of incremental build integration, the adapted algorithm is integrated into the static analysis platform Magellan [49]. The evaluation compares RTA call graph precision to an implementation of CHA and evaluates the incremental runtime of the prototypical implementation.

4.2.2 Rapid Type Analysis

The Rapid Type Analysis (RTA) [13] is an interprocedural call graph analysis that refines the program virtual call graph (PVG). The analysis constructs a conservative set for reachable methods and reachable type instantiations of a program in context of its entry points. The key difference between the CHA and the RTA is the concept of liveness. Methods, type instantiations, call sites and call instances that are reachable under the RTA are called *live*. Methods and types that are not considered live and therefore are unreachable are pruned from the call graph. This reduces the size of the call graph and increases its precision. The following definitions for liveness are used:

- A **method** is live, if it is a start method or if it is reachable through a live call instance.
- A **type** is live, if it is instantiated in a live method.
- A **call site** is live, if at least one of its call instances is live.
- A **call instance** is live, if the method it belongs to is live and it either is a direct call instance or one of its possible dynamic receiver types is live.

Algorithm 1 Modified Rapid Type Analysis algorithm

```
1: procedure rapidTypeAnalysis(M, T, S, I, M_S)
2:     Q_V ← ∅
3:     T_L ← M_L ← ∅
4:     for all m ∈ M_S do
5:         analyze(m)

6: procedure analyze(m ∈ M)
7:     if m ∈ M_L then
8:         return
9:     M_L ← M_L ∪ {m}
10:    for all s ∈ S, m' ∈ M, P ∈ 2^T : ⟨s, m, m', P⟩ ∈ I do
11:        Let i = ⟨s, m, m', P⟩
12:        if i ∈ I_D or (i ∈ I_V and T_L ∩ P ≠ ∅) then
13:            addCall(i)
14:        else
15:            addVirtualMapping(P, i)

16: procedure addCall(i ∈ I)
17:    Let ⟨s, m, m', P⟩ = i
18:    if i ∈ I_I then
19:        instantiate(type(m'))
20:    analyze(m')

21: procedure instantiate(t ∈ T)
22:    if t ∈ T_L then
23:        return
24:    T_L ← T_L ∪ {t}
25:    for all i ∈ I : ⟨t, i⟩ ∈ Q_V do
26:        Q_V ← Q_V \ {⟨t, i⟩}
27:        addCall(i)

28: procedure addVirtualMapping(P ∈ 2^T, i ∈ I)
29:    for all t ∈ P do
30:        Q_V ← Q_V ∪ {⟨t, i⟩}
```

Algorithm 1 on the preceding page shows the RTA algorithm from Bacon [13, p. 47]. The notation is slightly modified to fit the notations used throughout this chapter. The input comprises the PVG (the sets of concrete methods M, concrete types T, call sites S and call instances I), and the set M_S of start methods of the program. The virtual mapping Q_V maps a type t to the virtual call instances, that use the type as static receiver type.

ANALYZE is used as starting point of the analysis. It is called for each start method (see Line 5). The ANALYZE procedure is called for each reached method. The whole analysis process works recursively, by traversing the call instances of a method (Line 10), analyzing each call instance (Line 13), and further analyzing the target method of each call instance (Line 20). Each virtual call instance that is not be immediately resolved, i.e., for which live receiver types do not exist, is stored in the virtual mapping Q_V (Line 15). If, at a later point of the analysis, one of the receiver types is instantiated, the call instance can be retrieved from the map. The set of live types T_L is constructed by the INSTANTIATE procedure at Line 21. A type becomes live, if the constructor of that type is the target of an instantiating call instance, that becomes reachable (Line 19). A repeated analysis of methods and types that already are recognized as being live is avoided by the guard conditions at Lines 7 and 22.

Although Bacon [13] states that "there are no special considerations for constructing the PVG for Java", there nevertheless are Java features, that lead to control flow that is not recognized using the algorithm described above. In this section, custom mechanisms for static initializers and finalizer are presented. Also a generic configuration mechanism is shown that is used to incorporate control flow for calls to methods outside the program.

Static Initializers

A static initializer is a method that is used to initialize the static fields of a class or interface. A class or interface has exactly one static initializer if it declares at least one static field or if the static initializer is explicitly defined. The static initializer is called once by the Virtual Machine, when the class it belongs to is initialized. Before the call, the static initializers of all superclasses and of implemented or extended interfaces are called, if not called already [92, §2.17.4].

Since the static initializer is called by the Virtual Machine and not by a statement from another method, the control flow to a static initializer has to be approximated as follows: For each source element that may cause a static initializer call, a special direct call site to the static initializer is added. This call site is only reachable if the surrounding method of its corresponding

source element is reachable. The static initializer itself is also expanded with a special call site pointing to the static initializer of its superclass. If a superclass (or superinterface) has no static initializer, the next available static initializer in the superclass and interface hierarchy is used.

Finalizers

A finalizer is a method that is called by the Virtual Machine at some execution point before the object it belongs to is claimed by the garbage collector [92, §2.17.7]. If finalizers are ignored during PVG construction, the control flow to the finalizers is not recognized, and type instantiations in finalizers will not cause the instantiated type to be considered live.

To reflect the corresponding control flow in the PVG, a special finalizer call site is added to each instantiating call site. If a class does not define a finalizer, the next available finalizer in the superclass hierarchy is called. To summarize: the finalizer is reachable, if the instantiating call site is reachable.

Simulating control flow

There are cases, where the control flow is not explicit in the source code. The following cases are examples for implicit control flow that are not recognized using the call graph construction as described above:

- callbacks via native methods, for example reactions to mouse clicks,
- method calls by the virtual machine, like the invocation of Thread.run() after calling Thread.start()
- methods invoked via *Java Reflection API*, which includes the invocation of methods of dynamically loaded classes. Livshits [94] shows an approach to deal with reflection using a points–to analysis. The presented approach still needs user configuration to specify control flow relations that rely on the dynamic input of the program. As an implementation of a points to analysis for Magellan remains still to be done, the focus is on the configuration mechanism.

Configurations are used to account for the liveness of types reached through implicit control flow. The mechanism used is detailed in Section 4.4.4.

4.3 Automatic Incrementalization

This section describes the automatic incrementalization of the RTA using the Datalog engine described in Section 2.3.

```
1  pvgstatic(From,To):−
2    (def(From,_,_,_,_,invokeFunc(To,_,_))|
3    invokeProc(From,_,_,To,_,_)),
4    (private(To)|static(To)).
5
6  pvgdyncall(From,Ts):−
7    (def(From,_,_,_,_,invokeFunc(Ts,_,_))|
8    invokeProc(From,_,_,Ts,_,_)),
9    (protected(Ts)|public(Ts)|default(Ts)).
10
11 pvgdyncall(From,To):−
12   method(To,Name,ChildClassId,Par,RVal),
13   method(Ts,Name,ClassId,Par,RVal),
14   inherits(ChildClassId,ClassId),
15   pvgdyncall(From,Ts).
16
17 pvgcall(From,To):−
18   pvgstatic(From,To)|pvgdyncall(From,To).
```

Listing 4.2: PVG Construction

To achieve automatic incrementalization using Datalog, the source representation as described in Section 2.3.2 is used. For the RTA, types, methods and method calls are modeled as Datalog facts. Further the inheritance relationship is encoded.

Based on this program representation, the PVG is constructed as described in Listing 4.2. pvgcall/2 (Line 17) relates a method From to a method To, if there is a call in From, whose target may be bound to To. This is the case if there is either a static call or a dynamic call between the two methods. Static calls are found using pvgstatic(From,To) (see Line 1). Virtual calls are found using pvgdyncall(From,Ts) (defined in Lines 6 to 15). A virtual call is either an invokeFunc/3 or an invokeProc/6 and its target is not private. Line 11 relates the caller to all concrete visible methods in classes that inherit from the target type of the call. This is necessary, because otherwise, only the static target of the call would be found and not the calls to its children.

Listing 4.3 on the facing page shows the definition for the RTA, which closely follows the definition from Section 4.2.2. Line 1 shows an example, how start methods are defined. Here, all methods with the name 'main' are selected as start methods. Line 4 shows the definition of live methods, being either start methods or methods in classes that are called from live methods.

```
1  startMethod(A):−
2    method(A,'main',_,_,_).
3
4  livemethod(Id):−
5    startMethod(Id)|(
6    liveinstance(From,Id),
7    livemethod(From)).
8
9  livetype(Id):−
10   livemethod(Method),
11   instantiated(Method,Id).
12
13 instantiated(From,Id):−
14   invokeProc(From,_,_,To,_,_),
15   method(To,'<init>',Id,_,_).
16
17 liveinstance(From,To):−
18   (pvgstatic(From,To)|
19   (livetype(Type),
20    method(To,_,Type,_,_),
21    pvgdyncall(From,To))),
22   livemethod(From).
```

Listing 4.3: RTA Construction

Live types (Line 9) are instantiated (Line 13) from live methods. And finally, a live call instance (Line 17) is a call from a live method to a live type.

4.4 Manual Incrementalization

The implementation discussed in this section is based on the diploma thesis by Michael Achenbach [1]

This section describes the manually incrementalized implementation of the rapid type analysis. In the next section, an overview over the approach to incrementalization is given. Section 4.4.2 describes, how the incremental program virtual graph (PVG) is constructed. Section 4.4.2 presents the incremental rapid type analysis. It is shown, how to incrementally update the live values of the call graph, based on the PVG and Δ_{PVG}. In Section 4.4.4, the integration into Magellan, the platform for static analysis is detailed.

4.4.1 Overview About the Incremental Process

Algorithm 2 General incremental process

1: Input:
2: $\mathcal{P} \leftarrow$ input program
3: $G_{CHG} \leftarrow$ class hierarchy graph of $\mathcal{P}$
4: $G_{PVG} \leftarrow$ initial PVG construction using $\mathcal{P}$ and G_{CHG}
5: $G_{RTA} \leftarrow$ applying initial RTA algorithm to G_{PVG}
6: **loop**
7: $\quad \Delta_{\mathcal{P}} \leftarrow$ program delta
8: $\quad \Delta_{CHG} \leftarrow$ hierarchy graph delta
9: $\quad G_{PVG}, \Delta_{PVG} \leftarrow$ incremental PVG construction using $\Delta_{\mathcal{P}}$ and Δ_{CHG}
10: $\quad G_{RTA} \leftarrow$ incremental RTA update using Δ_{PVG}

An overview of the incremental approach is shown in Algorithm 2. Given a program $\mathcal{P}$ and its class hierarchy graph G_{CHG}, the call graph is initially constructed according to CHA. The call graph is then transformed by applying the RTA (i.e. the liveness of types and methods is determined). For each program modification (e.g., after a developer saves a program artifact), Lines 7–10 of Algorithm 2 are performed, where first the program delta and then the type hierarchy delta is computed. Then the PVG is updated using

the program delta Δ_P and the class hierarchy delta Δ_{CHG}. During this step, the PVG delta Δ_{PVG} is calculated. This is described in detail in the next subsection.

4.4.2 Incremental Program Virtual Call Graph

The PVG is constructed using the CHA from an input program $\mathcal{P}$ and its Class Hierarchy Graph G_{CHG}. The PVG needs an incremental update if changes occur in $\mathcal{P}$ or CHG. In this subsection, data structures are described that allow an efficient navigation through the PVG. Then atomic modifications of $\mathcal{P}$ and G_{CHG}, that can influence the PVG, are identified. For each atomic modification an update rule is described that collects modified elements of the PVG for a re–analysis. Other basic algorithms that provide the input for the incremental PVG update are already provided by the analysis framework Magellan. These are the incrementally updated source representation and the type hierarchy update Δ_{CHG}.

Data structures enabling efficient navigation in the PVG

In the callgraph as described above, calling methods contain a set of call sites comprising call instances, each pointing to the target method of the call (callee). The graph traversal from a caller to a callee is thus very fast. But the retrieval of the set of callers of a particular callee is very difficult. As fast navigation between the data structures in the call graph is necessary, the relationship between them is kept up to date with the following maps[3].

Method Map: Methods are mapped to all possible call instances. This allows to retrieve all callers of a method.

Instantiating Map: Types are mapped to all instantiating call instances.

Access Map: Methods are mapped to a list of possibly overriding methods.

Virtual Map: Receiver types are mapped to all possible call instances (in addition to the mapping for the non-incremental RTA, where receiver types are mapped to reached call instances).

With an implementation of these mappings dependencies of incremental program modifications can be calculated faster. For example, when deleting a method, the caller of this method can easily be retrieved and updated.

[3]The optimizations in this and the later sections are geared towards object-oriented runtime environments, as the implementations are done in Java.

Integrating program changes into the PVG

The following list itemizes the changes to Java code that influence the PVG and describes the rules that are applied to the PVG to integrate the changes.

Type marked abstract: The type is removed from all receiver type sets of all virtual call instances, because it can no longer be instantiated.

Type made concrete: The concrete type is added to the sets of possible receiver types of all call instances pointing to methods which are declared in supertypes of type t and type t itself. This is done using the method map, which maps types to all possible call instances.

Type change: If a type t changes, all call sites, that use t or a supertype of t as static receiver type, need an update because the receiver type sets of the call sites calling a method in t or a supertype of t may change. The call hierarchy graph delta Δ_{CHG} is used to retrieve all supertypes of t in the old and in the new program. Then the method map is used to retrieve all call sites calling one of these types. The call instances that belong to one of these call sites are recalculated.

Add method implementation: If a method m containing a code body is added, the corresponding call instances must be created for each call site calling m. Also, all call sites and call instances that contain m as a source method must be recalculated.

Remove method implementation: If a method is removed, the call instances and call sites, that belong to the removed method, must be deleted. This is done using the method map.

Add/remove virtual method: If a virtual method[4] is added to or removed from a class, then all call sites that could be target of a method dispatch instead of the changed method must be recalculated. These are call sites in super classes, in the class itself and in methods in subtypes that contain a call instance pointing to methods that have a signature matching the added/remove method. For the changed method m, the set of supertypes of the class in which m is declared is retrieved from the type hierarchy. For each supertype, the access map is used to retrieve the set of visible methods corresponding to the signature of m. For each visible method, the method map is used to retrieve the

[4] A virtual method is a method that is subject to dynamic dispatch, i.e. a method that is not static, not private and not a constructor method.

affected methods, whose content needs to be recalculated. These methods contain a call site that is represented by the visible method. If a virtual method is added or removed, this is also considered an addition or removal of a method implementation. Thus the corresponding rules are also applied.

Change modifiers of a virtual method: Changing the abstract modifier of a virtual method is treated like the addition resp. removal of a virtual method, because only non–abstract methods can be called. If the modifier of a virtual method m is changed, this is also considered a change of a virtual method and the corresponding rules are applied.

Add/remove method with more precise signature: If a method m (including constructor and interface methods) is added and a method m' with a less precise signature already exists in the program, all call sites, whose static receiver type and signature match the method with the less precise signature, need an update. If method m is removed and a method m' with a less precise signature remains in the program, all call sites, whose static receiver type and signature match method m need an update.

Add/remove static initializer/finalizer: If a static initializer or finalizer is added or removed, each method that contains call sites or field accesses[5] potentially causing a call to the added method, need an update.

For the addition of a static initializer, call sites must be added to all methods calling the constructors and static methods of the changed type. To retrieve these call sites, the method map is used.

For the addition of a finalizer to a type t, the type hierarchy is used to retrieve the set of all subtypes of t (including t). For each type in this set, the instantiating map is used to retrieve the instantiating call instances that may need a finalizer call instance. The call instances then are updated accordingly.

If a static initializer or finalizer is removed, all call sites, that point to the removed method, need an update. The method map is used to remove the call instances pointing to the removed method.

[5]The impact of static field access (JVMSPEC §2.17.4, see [92]) is currently not taken into account.

Delta of the Program Virtual Call Graph

Algorithms that build upon the PVG need access to the changes of the program virtual call graph Δ_{PVG} after a build process. The structure comprises the following sets:

- modified, added, and removed methods (M_m, M_a, and M_r). A method is considered modified, if it is added or one of its call sites is added, removed, or modified by one of the atomic modifications described above.
- added and removed start methods (M_{Sa} and M_{Sr})
- modified, added, and removed call instances (I_m, I_a, and I_r). The sets I_a and I_r reflect the addition and removal of call instances corresponding to the atomic modifications above. A call instance is considered modified, if it is neither added nor removed, but its set of receiver types is modified.

Optimizations

The control flow to methods of java.lang.Object is omitted, because no further control flow is caused by them, as Object cannot call back into application code. For example, the Java specification requests a call to the constructor of Object from each constructor of a type, that extends Object. If each such edge would be inserted into the call graph, the method map would have to be updated for each modified call instance pointing to the constructor of Object. That would increase runtime without gaining more precision, as the edges to methods in Object can be calculated on demand, if necessary for a further analysis.

4.4.3 Incremental Rapid Type Analysis

This section describes the modifications to the RTA that are necessary to integrate it into the incremental build process and to compute the correct reachability of methods and types from the previous graph and the delta of the PVG. First, the notion of distance in this context is discussed. Then, the algorithm for the incremental RTA is presented. The full build case is described as a special case of the incremental build. Optimizations are discussed at the end of the section.

Distance from Start Methods to Method Nodes in the Call Graph

The distance between two nodes in a graph is the number of edges in a shortest path connecting them. The distance in the call graph represents the distance to the start methods according to the breadth-first search strategy. The distance value is modeled as a field at each method, type, and call instance for fast access. For each start method, the distance is 0. The distances for methods and types is one plus the minimum of all methods that call the method or instantiate the type. The distance of methods and types that are not live (and therefore not reachable from a start method), is ∞.

When $d(x)$ and $d_{calc}(x)$ are used in the following, $d(x)$ is used as a variable, that can be modified and queried in constant time and $d_{calc}(x)$ is used as a function, that calculates the distance of x from its predecessors in $\mathcal{O}(|P_x|)$ time, where $|P_x|$ is the number of direct predecessors of x.

Algorithm Description

An overview of the incremental rapid type analysis (IRTA) is shown in Algorithm 3 on page 95. The IRTA labels all methods, types and call instances with their distance to the start methods. Using a breadth-first strategy, the PVG is traversed starting at the modified methods, types, and call instances with the smallest distance.

In the RTA algorithm presented in Section 4.2.2 a depth–first search (DFS) is used to traverse the call graph. Using a DFS for the IRTA would cause the following problems:

- The addition of a call instance i triggers the analysis of the complete part of the program that is reached through i, before any other call site or call instance in the method of i is analyzed.
- In a call graph, often more than one path connects two methods. If a path between nodes is interrupted due to an incremental modification, it is expensive to determine whether another path exists.

To tackle the second problem, the weight of the reachability of a node can be stored locally in a way that is similar to reference counting in garbage collection algorithms [83]. A reachability value v counts per method, how often it is reached. Every time, an edge is removed from the call graph, the value v of the target method is decremented. If v reaches zero, the method of v becomes unreachable.

This approach works fine, as long as there are no circles in the graph. But recursive and mutually recursive methods of the analyzed input program

cause circles in the call graph. Circles break reference counting, because each element in the circle has a reference to it even after the lag external reference is deleted. A requirement of the call graph in this work is, that it preserves the same precision over time. In garbage collection algorithms, this requirement is not very important. If some parts of the object graph can not be claimed due to circularity, a so called tracing garbage collection algorithm is performed from time to time. The object graph is analyzed, starting at some entry points (references on the stack, which compare to the start methods of the call graph). The reachable parts of the object graph are marked and the remains is claimed by the garbage collector [83].

Using breadth-first search (BFS) instead of DFS for the IRTA-algorithm, solves both problems mentioned above. The graph can be traversed using depth ordering: To decide the reachability of a node, only edges from nodes with smaller distances are used. Furthermore, the calculation of components in the same iteration is parallelizable, which increases the performance of the analysis on multi-threaded architectures. As a result of the BFS, the methods are marked with their distance to the start methods.

The input of the IRTA is the updated program virtual-call graph G_{PVG} and its modification delta Δ_{PVG}, that are the output of the incremental PVG construction. G_{RTA} contains the IRTA information (e.g., the live values of methods and types).

The modification extent E (Line 5) contains the methods and types, that have to be re–analyzed. Δ_{PVG} is used to initialize E with the following methods (see Algorithm 3, Line 6):

- added start methods,
- live methods that were modified since the last build,
- live methods that may not be reached anymore, because a call instance that points to them is removed.

Modified methods that are not live do not have to be part of this set. If they are reached later, their liveness is updated then. Initially, no types and call instances are in the re–analysis set.

Lines 9–17 are repeated until the extent E contains no more methods and types with a distance bigger than d. Each loop iteration corresponds to an exploration step of the breadth-first search strategy.

In contrast to the BFS shown in [39] the IRTA uses two frontier sets that are explored in lockstep.

- The first frontier is called the *invalidation frontier*. During exploration, methods, types, and call instances may not be reached anymore, due

Algorithm 3 Incremental Rapid Type Analysis

```
1: Input:
2: G_PVG ← the updated PVG structure
3: Δ_PVG ← the modification difference of the PVG
4: G_RTA ← liveness and other information regarding the RTA
5: E ← the extent, calculated from Δ_PVG, contains the methods
       and types, that have to be analyzed and the methods
       and types, that may not be reached anymore
6: E ← added start methods (marked with distance 0)
7: d ← smallest distance of methods and types in E
8: while E_≥d ≠ ∅ do
9:     F_R ← methods and types in E with distance d, that have to
           be re–analyzed (re–analysis frontier)
10:    for all m ∈ E_d do
11:        ANALYZE(m,d)
12:    F_V ← methods and types in E with distance d, that may not be
           reached with distance d anymore (invalidation frontier)
13:    << explore the re–analysis frontier F_R (see Algorithm 4)>>
14:    E ← E ∪ newly reached methods and types from F_R (mark with
           distance d + 1)
15:    << invalidate methods and types in the invalidation frontier F_V >>
16:    E ← E ∪ methods and types from F_V, that may not be reached
           anymore
17:    d ← d + 1
```

to removed edges or nodes in the PVG. The unreachable elements are identified and marked as not live.

- The second frontier is called *re–analysis frontier*. During exploration newly reached methods and types are discovered and marked live markings. The distance values of reached methods and types are updated.

Every time the BFS loop expands to the next frontier and the distance value d is incremented, each method, type, or call instance with a distance smaller than d must be already analyzed correctly. This ensures the correctness of every further distance calculation.

Both frontiers are explored together: the re–analysis frontier F_R (see Line 9) and the invalidation frontier F_V (see Line 12). The methods called from the explored methods are put into the extent (Line 14), together with the methods that are no longer reached with the current distance (Line 16). Lastly, the distance to explore is increased (Line 16).

Next, the exploration phases for the re–analysis frontier (Line 13) and the invalidation frontier (Line 15) are described in detail.

The **re–analysis frontier** contains methods, types and call instances, that have to be re–analyzed. Algorithm 4 on the facing page describes the exploration of the re–analysis frontier with a particular distance d.

When a method is analyzed (Line 1), each call instance of the methods code body is checked, if the call instance is a direct call instance, its distance is smaller or equals the BFS distance, or a recalculation of the distance matches the BFS distance (Line 4). If all three conditions hold, the call instance is analyzed with the addCall procedure (Line 32).

A call instance may have a smaller distance than its currently analyzed source method m if m had the smaller distance in a former build and was invalidated in the current build. Then it may now be reached again with a bigger distance. If the distance of a virtual call instance i is greater than d, the smallest distance of the receiver types of i must be greater than d. In that case, the call instance has to be re–analyzed later, when the BFS reaches that distance.[6]

When a type t is instantiated (Line 9), each call instance, that uses t as receiver type and whose source method is reached, is directly analyzed, if the distance of the source method is smaller than the BFS distance. Otherwise, the source method's reachability is unclear and the call instance has to be re–analyzed later.

[6]If no receiver type is live at that point, $d_{calc}(i)$ returns ∞ and the call instance will not be analyzed.

Algorithm 4 Exploring the re–analysis frontier (part 1)

```
1: procedure ANALYZE(m ∈ M, d ∈ ℕ)
2:     for all s ∈ S, m′ ∈ M, P ∈ 2^T : ⟨s, m, m′, P⟩ ∈ I do
3:         Let i = ⟨s, m, m′, P⟩
4:         if i ∈ I_D or d(i) ≤ d or d_calc(i) = d then
5:             addCall(i, d)
6:         else
7:             d(i) ← d_calc(i)
8:             R ← R ∪ {i}

9: procedure INSTANTIATE(t ∈ T, d ∈ ℕ)
10:    for all i ∈ I : ⟨t, i⟩ ∈ Q_V do
11:        Let ⟨s, m, m′, P⟩ = i
12:        if m ∉ M_L then
13:            continue
14:        if d(m) ≤ d then
15:            addCall(i, d)
16:        else
17:            d(i) ← d(m)
18:            R ← R ∪ {i}

19: procedure analyzeInstance(i ∈ I_V, d ∈ ℕ)
20:    Let ⟨s, m, m′, P⟩ = i
21:    if d(m) > d then
22:        d(i) ← ∞
23:        return
24:    if d_calc(i) = d then
25:        addCall(i, d)
26:    else if d_calc(i) < ∞ then
27:        d(i) ← d_calc(i)
28:        R ← R ∪ {i}
29:    else
30:        d(i) ← ∞
31:        R ← R\{i}
```

Algorithm 4 Exploring the re-analysis frontier (part 2)

```
32: procedure ADDCALL(i ∈ I, d ∈ ℕ)
33:     Let ⟨s, m, m′, P⟩ = i
34:     I_L ← I_L ∪ {i}
35:     if i ∈ I_I then
36:         Let t = type(m′)
37:         if d(t) > d + 1 then
38:             d(t) ← d + 1
39:         R ← R ∪ {t}
40:     if d(m′) > d + 1 then
41:         d(m′) ← d + 1
42:     R ← R ∪ {m′}
```

When a call instance i is re–analyzed at Line 19 of Algorithm 4 on the previous page, its distance is reset at Line 22 if its source method's distance is greater than the actual distance. If the calculated distance matches the BFS distance, ADDCALL for the call instance is called immediately. If its distance is bigger than d but smaller than ∞, i.e., the call instance is still reachable, i is scheduled for a later re–analysis. Otherwise, the call instance cannot be reached.

The ADDCALL procedure for call instances is shown at Line 32 of Algorithm 4 on the preceding page. The newly reached methods and types are added to the re–analysis set $\mathcal{R}$ to be explored in the succeeding BFS iteration if the recalculation criterion holds. The distance of newly reached methods and types is set to $d + 1$.

The exploration of the re–analysis frontier can discover methods and types with distance $d+1$ in the extent E, whose reachability was unclear, i.e., which were part of the invalidation frontier F_V in the previous BFS iteration.

If a method or type x is explored with distance d (see Algorithm 3 on page 95, Line 14), and was not analyzed before in the same incremental build, it has to be re–analyzed. If $d(x)$ is greater than the distance of the current exploration, x is part of the set of methods to be re–analyzed ($\mathcal{R}$) or part of the set of methods that are candidates for invalidation ($\mathcal{V}$). Then, x is marked live and is removed from $\mathcal{V}$ and its distance is updated. x is added to set $\mathcal{R}$, so that its call instances are analyzed in the following iteration.

The **invalidation frontier** contains methods and types, that were reachable in a former build, but may not be reachable anymore.

During the exploration of the re–analysis frontier, every reached compo-

nent is removed from the invalidation set. The remaining methods and types, that were reached before with the BFS distance d, are now unreachable with that distance, but may be reachable with a greater distance. Therefore they are marked as not live and their distance is reset to infinity. The predecessors of x are added to the re–analysis set. The predecessors of a method are the call instances pointing to the method. The predecessors of a type are the call instances, that instantiate the type. The predecessor of a call instance is its source method. A virtual call instance has additionally its possible dynamic receiver types as predecessors. If some of the predecessors are still reachable with a bigger distance or become reachable again (as will be recognized in a later BFS iteration), x will be reached and re–analyzed, too. Dependent methods and types will be scheduled for invalidation if x caused their reachability and itself is not reachable.

If a method m is deleted or becomes unreachable with its former distance, it is marked as not live. Each call instance of the method is marked as not live, its old distance is stored, and its distance is reset. If the call instance i was the reason for the reachability of the method m, i.e., if $d_{old}(i)+1 < d(m)$ and no other call instances with the same or smaller distance still point to method m, m is added to the invalidation set $\mathcal{V}$. If i needs to be re–analysed and is an instantiating call instance, the type t, that is instantiated by i, has to be checked, too.

Each type t in the invalidation set $\mathcal{V}$, is marked as not live, its old distance is stored, while its current distance is reset ($d_{old}(t) \leftarrow d(t)$, $d(t) \leftarrow \infty$). Using the virtual map, for each call instance i that uses type t as static receiver type the following three steps are taken:

1. If type t cannot have caused the reachability of call instance i (because $d_{old}(t) > d(i)$), the call instance is ignored. Otherwise, the distance of i is updated ($d(i) \leftarrow d_{calc}(i)$).

2. If the type actually caused the reachability of i (because $\max(d(m), d_{old}(t)) < d_{calc}(i)$), the call instance is marked as not live and added to the re–analysis set $\mathcal{R}$.

3. If the call instance i caused the reachability of a target method m' (because $d_{calc}(m') > d(m')$), m' is added to the invalidation set $\mathcal{V}$.

Initial Build

The initial construction of the call graph (see Algorithm 2 on page 88, Line 5) is a special case of incremental building. The extent E contains only the added start methods M_{Sa}. Because the call graph is still unexplored, the

invalidation processing can be omitted. If a full build is necessary after an incremental build, the values stored at each reachable component (e.g., distance or several flags) have to be reset by traversing the call graph.

Optimizations

A repeated analysis of methods, types, or call instances can be avoided by tagging elements as already analysed. If a binary flag would be used for this purpose, it would need to be reset for each build. Therefore, a unique id is assigned to each incremental build. Each method, type, or call instance is tagged with this id, when it is analyzed. The id then is queried to avoid a repeated analysis.

As in the RTA algorithm (see Section 4.2.2), the membership of components in global sets is modeled as flags at each component, to ensure constant time for membership tests and add operations. Also the receiver map, the method map, and the call instance map are implemented directly at the components. For example, a method should store the list of predecessor call instances mapped with the method map directly in its structure. So, access to the mapped elements can be achieved in constant time.

To avoid unnecessary recalculations of distance values, e.g., when checking for the invalidation, a reachability counter is used. Each method, type or call instance x counts the number of times it is reached with the smallest distance. If a predecessor of x with that distance is removed or becomes unreachable, the distance counter is decremented. If it reaches 0, x is not reachable with that distance anymore.

To achieve fast access to the methods, types and call instances in the extent E with a particular distance d (see Algorithm 3 on page 95, Line 9), the extends are implemented as arrays of lists. Each entry in the array carries elements with a distance equal to the array index. This does not cause space problems, since the maximum distance is expected to be much smaller than, for instance, the number of methods in the program.[7] Multiple insertions of elements to the lists are avoided by using the flags at each element.

4.4.4 Integration into Magellan

This section describes the implemented analyses and their integration into Magellan. The analyses of Magellan are producer–consumer units, that can be executed sequentially or in parallel (see Section 2.2.1). Each analysis pro-

[7] In preliminary studies, the maximum distance of a call graph with ca. 4000 methods was 25.

duces a so called *fact*, that contains the analysis results, which is consumed by subsequent analyses.

The necessary base analyses are already provided by Magellan. These are the CHG analysis, the CHG changed analysis, and analyses, that parse the class files and provide the quadruples representation of the Java bytecode.

The auxiliary analyses described next preprocess the IRTA input and increase performance by caching data structures.

Configuration Analysis

The configuration analysis provides additional configuration information for other analyses. It generates a fact from an XML configuration file, which is provided by the user of the analysis. In the current implementation, regular expressions can be defined, that match sets of class members.

A difference structure is calculated incrementally, when the developer modifies the configuration. The difference structure provides the added and removed patterns for subsequent analyses.

Start Method Configuration

This analysis is used to collect start methods for the call graph analysis by matching the patterns provided by the configuration analysis against the methods in the analyzed program. The pattern analysis consumes the output of the configuration analysis and calculates a set of class members, that match the pattern configuration. It uses the document fact related to class files and analyzes the class members. It works incrementally for both input sources. If the pattern set changes, the class files are re–analyzed and the set of class members is updated. If a class file is modified, only the modified file is re–analyzed. A difference structure, that provides the added and removed class members for a certain property, is calculated incrementally.

Simulating Control Flow Through Configuration

This analysis consumes the output of the configuration analysis and creates artificial call sites to simulate implicit control flow.

Special call sites are generated from a configuration file. A special start method m_{sim} is added to the set of start methods M_S. Then, patterns from a configuration file for generating simulated call sites are parsed. Each pattern contains the fully qualified name of the static receiver type of the call, a name, and a signature. A flag is used to choose between a direct or virtual call site. The example shown in Listing 4.4 on the following page shows a pattern, that describes a virtual call to java.lang.Runnable.run():

```
|<"java.lang.Runnable", "run", "()V", VIRTUAL>
```

Listing 4.4: jEdit configuration file excerpt

The patterns for artificial call sites are analyzed and wildcards are resolved by matching the pattern against possible method execution in the analyzed source program. The set of simulated virtual and direct call sites is added to the set of call sites of m_{sim}. The simulated call sites are not added to the call graph—they only cause the reachability of their target methods. A control flow edge from the calling method to the called method is not part of the call graph. This has to be taken into account, when constructing a data–flow analysis that builds on the call graph. E.g., a method call via Reflection API from method m to method m' yields a return value, that influences the data–flow of method m. The approach discussed above only constructs an edge from m_{sim} to m' and thus ensures the liveness of m', without constructing the edge from m to m'.

The start method is parsed during PVG construction. The special call sites are interpreted like other *virtual* and *direct* call sites of the program.

This approach is also used for interface–only library analysis—if only the interfaces of dependent libraries are analyzed, some control flow is omitted, e.g., a call to the equals() method of a subtype of java.util.Set may refer to equals() methods in user defined source code. The equals() method then is marked as live with a corresponding pattern in the configuration file.

Fast Method Access Analysis

In the current data model of Magellan, method declarations of a class or interface are stored in an array per type. Searching a method with a particular name has a linear time complexity over the size of the array. This was a design decision in Magellan to create a compact data representation even at the price of sacrificing performance for cases with frequent searches. Because the calculation of override and inherit sets often accesses a particular method, a caching of methods increases performance.

The fast method access analysis produces a fact that caches the class-method relation to reduce the runtime of a method lookup. The analysis provides two hash maps; one that maps from class files to method names and one that maps from method names to the list of methods, matching the name (whose number is expected to be very small in practice). The mapping is incrementally updated, if the underlying class file facts are modified. The usage of this analysis increases the overall performance approximately by a factor of 3.

Program Delta Analysis

The data model of modified files in Eclipse is quite coarse. If a developer modifies a Java file and saves it, the complete file is recompiled. The corresponding class file is first removed, and the newly compiled class file is added to the resource base. Magellan directly uses a set of removed and added class files in the data model to express an incremental modification. The program delta analysis transforms the data structures to provide a finer level of granularity for the incremental PVG construction. The output granularity of this analysis equals the atomic program modifications, described in Section 4.4.2.

The program delta is calculated for each incremental modification. It contains type deltas for each modified type. A type delta contains the information about added or removed *abstract* modifiers of a type, and about added or removed types. Each type delta also contains a set of method deltas. A method delta contains the modification information of the *abstract*, *static* and visibility modifiers of a method and whether a method is added, removed, or whether its content is modified.

4.5 Comparison of the Approaches

This section compares the two implementations. After describing the evaluation setup in the next section, Section 4.5.2 presents the measurements for the automatically incrementalized implementation. Section 4.5.3 details the measurements for the manually incrementalized implementation. In Section 4.5.4 conclusions are drawn.

4.5.1 Setup

For the evaluation of the implemented algorithms the open source editor jEdit[8] was chosen as the program to analyze. The sources contain 872 type declarations, 5,407 method declarations and 39,990 call sites. Also jEdit uses reflection extensively. It uses the BeanShell[9] Java interpreter and scripting environment, which itself is written in Java and dynamically executes Java source code. Some of jEdit's GUI elements are instantiated using BeanShell scripts. Listing 4.5 on the next page shows a configuration file for GUI components, that are instantiated via BeanShell. There, the class LogViewer is instantiated in Line 5 and the class VFSBrowser is instantiated in Line 8. Using reflection, other classes are dynamically instantiated from property

[8]Version 4.2 was used from `http://www.jedit.org`.
[9]See `http://www.beanshell.org`.

```
1  <!DOCTYPE DOCKABLES SYSTEM "dockables.dtd">
2
3  <DOCKABLES>
4    <DOCKABLE NAME="log-viewer">
5      new LogViewer();
6    </DOCKABLE>
7    <DOCKABLE NAME="vfs.browser">
8      new VFSBrowser(view,position);
9    </DOCKABLE>
10   ...
11 </DOCKABLES>
```

Listing 4.5: Dockable windows configuration (excerpt from jEdit source)

```
1 #{{{ Shortcuts pane
2 options.shortcuts.label=Shortcuts
3 options.shortcuts.code=new ShortcutsOptionPane();
4 options.shortcuts.select.label=Edit Shortcuts:
5 #}}}
```

Listing 4.6: Property configuration (excerpt from jEdit source)

configuration files in jEdit. An excerpt of such a configuration is shown in Listing 4.6, where a ShortcutsOptionPaneis instantiated in Line 3. These instantiations are not present in the source code, but nevertheless must be taken into account, because the precision of the RTA mostly depends on class instantiations.

The evaluation was done using a 2 GHz Intel Core2Duo workstation with 3.2 GB RAM. The performance measurements were applied in a JUnit environment, that executes the algorithms decoupled from Magellan and Eclipse, to avoid runtime and space overhead from Eclipse's GUI and worker threads. Each algorithm configuration is run three times and the best overall performance is chosen, to deal with variations caused by background processes.

The incremental build evaluation uses a set of program modifications to compare the incremental call graph update times with the runtime of the full build algorithm. Eight modifications show the atomic changes as described in Section 4.2.2. Four modifications show typical actions of an IDE user. For the evaluation of the incremental build, the simulated control flow and static initializer and finalizer handling are turned on.

Because of the coarse-grained implementation of the current program delta analysis, each source modification always causes the modification of all

call sites in all modified source files. That leads to an increased update time even if no call sites, but e.g., a comment is modified.

The following lists the modifications that execute atomic changes:

- m_1: A static call site is first added to (a), then removed from (r) a large class containing 500 call sites.
- m_2: A virtual method is first added to (a), then removed from (r) a small class with 2 supertypes.
- m_3: A hierarchy link (*extends*) is first removed from (r), then added to (a) a medium sized class with six supertypes.
- m_4: A virtual call site is first removed from (r), then added to (a) a large class containing 562 call sites.
- m_5: The *abstract* modifier is first added to (a), then removed from (r) a method in a small class.
- m_6: The *abstract* modifier is first added to (a), then removed from (r) a small class.
- m_7: A static initializer is first added to (a), then removed from (r) a small class.
- m_8: 415 different call sites are first removed from (r), then added to (a) a large and some small classes.

The modifications that focus on IDE user actions are as follows:

- m_9: A medium sized class is modified without influencing call graph related sources.
- m_{10}: A new class is created (a), then completely removed (r); it has 3 supertypes after creation.
- m_{11}: The Eclipse "rename type" refactoring is applied first to a small (s), then to a big (b) class; the latter causes a modification of 272 types.
- m_{12}: The Eclipse "rename method" refactoring is applied to a virtual method in a small (s) and a big (b) class; the latter causes a modification of 83 types.

Run	update source representation	update call graph
m_1^a	594 ms	16,695 ms
m_1^r	608 ms	15,530 ms
m_2^a	127 ms	10,275 ms
m_2^r	114 ms	9,996 ms
m_3^a	267 ms	14,912 ms
m_3^r	270 ms	14,146 ms
m_4^a	202 ms	15540 ms
m_4^r	194 ms	14,352 ms
m_5^a	114 ms	10,141 ms
m_5^r	109 ms	9,763 ms
m_6^a	195 ms	13,984 ms
m_6^r	194 ms	13,465 ms
m_7^a	103 ms	14,215 ms
m_7^r	111 ms	13,685 ms
m_8^a	908 ms	15,281 ms
m_8^r	1,014 ms	16,549 ms
m_9	523 ms	18,314 ms
m_{10}^a	29 ms	10,166 ms
m_{10}^r	109 ms	9,615 ms
m_{11}^a	401 ms	16,997 ms
m_{11}^r	23,567 ms	52,684 ms
m_{12}^a	1,341 ms	17,456 ms
m_{12}^r	11,210 ms	26,182 ms

Table 4.1: Incremental build timings

4.5.2 Automatic Incrementalization

To evaluate the implementation, the setup described above was used. All runs are analyzing only interfaces of libraries. In case of the full build, the analysis took 200 seconds to analyse the project. From this, 29 seconds were used to create the source representation and 171 seconds were used to compute the call graph.

Table 4.1 shows the results for the incremental changes as described in the previous section. The first column shows, which change was measured. The changes are described in the previous section. The second column shows the time needed to update the source representation for the change. The third column shows the time needed to update the call graph.

The relative speedup from full build to incremental build of roughly one order of magnitude can also be seen here. But the incremental builds still take 10 s – 15 s. This is an order of magnitude to slow to be used in daily development work.

4.5.3 Manual Incrementalization

To evaluate the algorithms and the incremental approach, an empirical evaluation of precision and runtime was made. The first part of the evaluation compares the precision and soundness of call graphs obtained with different algorithm configurations. The handling of unrecognized control flow is examined. The second part of the evaluation details the incremental call graph update times.

Soundness

For the evaluation of precision and soundness, an analysis was used, that counts the number of unused methods. This so called *unused methods checker* was executed for each combination of algorithm type, Java specifics handling, and simulated control flow handling. No distinction was made between RTA and IRTA, because both algorithms provide the same call graph precision and differ only in incremental construction time.

The constructed PVG has 3,240 types, 28,476 methods, 39,990 call sites, and 85,982 call instances when analyzing only library interfaces. These numbers include the library types and methods. The number of live types, methods, and call instances depends on the used algorithm (CHA or RTA) and on the other configurations. If the used parts of libraries are also analyzed, the PVG contains 6,259 types, 58,182 methods, 206,722 call sites, and 819,205 call instances. The usage of RTA at least halves the size of the call graph in contrast to using CHA.

Table 4.2 on page 110 shows the analysis results for the given configurations. The first four columns describe the configuration for the runs (the runs are numbered in the rightmost column). The first column (lib) shows, if dependent libraries are fully analyzed (“yes”) or if only their interfaces are taken into account (“no”). The second column (algo) contains the name of the used algorithm (one of CHA, nRTA, or IRTA). The third column (spec) describes the handling of the Java specific “finalizers” and “static initializers” (“yes” or “no”). The fourth column (config) shows, if the simulated control flow configuration was used (“yes” or “no”).

The column named M_{unused} contains the number of methods that are deemed unused by the executed algorithm. To account for *false positives*

(FP_{min} in the next column), the set of unused methods as determined by the algorithm was compared to a set of methods that were used during a sequence of test runs. AspectJ [86] was used to log declaring class, name, and signature of executed methods. The resulting set contains 2585 methods, that were executed at runtime. Each method, that is marked as unused by the unused methods checker, but is logged by the runtime evaluation, is counted as false positive. The number of false positives is a lower bound, because the test runs have not logged all methods, that could possibly be executed. Using simulated control flow nearly halves the relative part of false positives (e.g., runs 1 and 4 or runs 5 and 8).

While the number of unused methods is halved, the number of false positives is reduced by approximately 75%. The simulation of static initializers and finalizers also increases precision. The more precise RTA call graph leads to more false positives. This is caused by method calls via reflection, that are not simulated, because the configuration patterns do not cover every possible call. Because of unrecognized class instantiations, more edges are missing in the RTA call graph than in the CHA call graph, which ignores class instantiation by definition.

Precision

To compare the RTA results, the constraint based definition of the Class Hierarchy Analysis (CHA) from [130] was implemented. This includes an analysis of the class hierarchy, as done in the PVG construction, and a reachability analysis, like that done for the RTA, but without taking class instantiations into account. To make the CHA work incremental, the following approach is implemented. The incremental PVG construction builds the base graph structure. Then, a reachability analysis calculates the live values of methods and types. The reachability analysis works like Algorithm 1 on page 83 but always adds the call instances of reached methods (see Line 12). A set of reached type instantiations is not constructed by the CHA. A virtual mapping from receiver types to call instances is not used. Revisiting call instances is not necessary, because type instantiations are ignored in CHA.

After an incremental modification, the live values are reset. Because the current implementation is not optimized for the CHA, only the evaluation of the reachability analysis is of interest.

Configuration

For each analysis run, all combinations of the following four parameters are used:

```
1 <"java.lang.Runnable", "run", "()V", VIRTUAL>
2 <"org.gjt.sp.jedit.options.*OptionPane", "<init>", "*", DIRECT>
3 ...
```

Listing 4.7: Simulation pattern configuration

1. One of the the algorithms CHA, normal RTA[10] (nRTA), and IRTA is used.

2. The handling of finalizers and static initializers can be switched on and off.

3. The same was done with the control flow simulation for unrecognized control flow. If activated, 11 manually designed configuration patterns are processed by the call graph configuration analysis and simulated call sites are created. An excerpt of the implemented patterns is shown in Listing 4.7. For the first pattern a virtual call site is created. When using the RTA, only those call instances become live, whose receiver type, that implements "Runnable", is instantiated. The second pattern causes the instantiation of all option panes. In jEdit the option panes are instantiated with the property configuration file shown in Listing 4.6 on page 104.

4. Furthermore, either the dependent parts of the library or only library interfaces are analyzed .

Full Build Evaluation

Table 4.3 on the following page shows the full build runtime. The parameters for static initializer/finalizer handling and simulated control flow are merged in the column "s&c", because the runtime of static initializer/finalizer handling showed not to be significant. The column t_{ext} shows the runtime of the preprocessing analyses. The next column (t_{build}) shows the runtime for construction of the PVG. The column headed t_Q shows the runtime of the PVG mapping update. The runtime of the reachability analysis is shown in the column named t_{reach}.

The initial build of the IRTA has some overhead compared to the nRTA. With growing call graph size the overheads relative size shrinks. The PVG

[10] In these runs, the PVG is computed incrementally, but the RTA is computed non–incrementally, i.e. from scratch based on the PVG.

setup				M_{unused}	FP_{min}		run
lib	algo	spec	config		total	%	
no	CHA	no	no	3030	588	19	1
			yes	1636	167	10	2
		yes	no	2936	582	19	3
			yes	1533	162	10	4
	nRTA & IRTA	no	no	4473	1055	23	5
			yes	2709	420	15	6
		yes	no	4252	977	22	7
			yes	2461	336	13	8
yes	CHA	no	no	1499	129	8	9
			yes	1331	46	3	10
		yes	no	1393	123	8	11
			yes	1225	41	3	12
	nRTA & IRTA	no	no	3049	480	15	13
			yes	2354	274	11	14
		yes	no	2787	389	13	15
			yes	2090	184	8	16

Table 4.2: Analysis results and false positives comparison

algorithm			performance units (in ms)				
lib	algo	s&c	t_{ext}	t_{build}	t_Q	t_{reach}	$\sum$
no	CHA	no	317.06	1231.47	40.36	3.53	1592.42
		yes	559.14	1224.26	49.39	8.54	1841.33
	nRTA	no	314.77	1278.11	37.60	6.77	1637.25
		yes	562.77	1208.13	49.65	22.16	1842.71
	IRTA	no	317.94	1306.30	36.67	12.62	1673.53
		yes	560.90	1220.38	48.84	42.25	1872.37
yes	CHA	no	1202.49	6417.07	419.82	57.31	8096.70
		yes	1560.74	6677.04	434.98	53.88	8726.65
	nRTA	no	1197.45	6372.37	417.25	111.87	8098.94
		yes	1559.09	6647.56	423.22	115.98	8745.85
	IRTA	no	1199.84	6357.02	422.00	165.19	8144.06
		yes	1413.78	6422.64	428.72	181.37	8446.52

Table 4.3: Full build performance

construction takes most of the time. The building and mapping update runtime of the PVG (t_{build} and t_Q) accumulate to $\approx 7s$ when analyzing libraries and when using control flow simulation. The corresponding nRTA runtime t_{reach} is $\approx 0.12s$. The analyses of source elements and the construction of the corresponding graph is much more time consuming than the fast traversal over the completed graph structure.

Incremental Build Evaluation

Table 4.4 on the next page and Table 4.5 on page 113 show the evaluation results for the atomic modifications. The labels of the modifications are superscripted with a or r to indicate, if the source elements are added (a) or removed (r). In the column headed t_{E1} the runtime of the PVG extent analysis is shown and t_{build} shows the build time for the PVG analysis. t_Q shows the runtime of the PVG mapping update. The reset action t_r is used by the nRTA and the incremental extent evaluation t_{E2} is used by the IRTA. Table 4.4 on the next page shows basic changes, where single call sites or methods are added or simple type hierarchy changes are made. The slower run for m_3 is caused by the PVG update. m_3 causes the update of all call sites, that refer to the six supertypes of the changed type. The identification process of those call sites is also very complex and causes a bigger extent calculation time t_{E1}. Table 4.5 on page 113 shows changes in the targets for virtual method dispatches.

Most of the runs have a good overall performance. The incremental update time of the IRTA ($t_{E2} + t_{reach}$) scales well and outperforms the update times of CHA and nRTA ($t_r + t_{reach}$) in most cases. The PVG update is faster than in the full build in most cases and the difference of the runtime of PVG update and reachability analysis is smaller than in the full build cases.

Table 4.6 on page 114 shows modifications, that focus on IDE user actions ($m_9 - m_{12}$).

The slow runtime of m_{11}^b in Table 4.6 on page 114 is obvious. Many other types and their methods have to be re–analyzed after renaming a class file with the Eclipse refactoring tool. This could be prevented if the refactoring semantics were available in the analysis. A rename action does not cause any changes, therefore, it could be ignored.

The overall incremental update runtime outperforms the full build runtime for all evaluated program modifications. Without an incremental algorithm, an update time of more than 8.7 seconds, when analyzing complete library dependencies, and of more than 1.8 seconds, when analyzing library interfaces, would be required for each incremental modification. This would slow down the feedback of subsequent analyses for an IDE user. Using the

setup			performance units (in ms)						
mod	lib	algo	t_{ext}	t_{E1}	t_r / t_{E2}	t_{build}	t_Q	t_{reach}	$\sum$
m_1^a	no	nRTA	1.23	10.01	5.06	24.66	22.07	13.55	76.59
		IRTA	2.54	9.84	1.24	24.54	21.60	0.62	60.38
	yes	nRTA	2.67	24.84	29.11	55.99	262.48	103.55	478.64
		IRTA	2.37	24.75	8.64	54.84	255.68	1.50	347.78
m_1^r	no	nRTA	1.24	7.38	5.14	25.76	21.32	13.73	74.58
		IRTA	1.14	7.64	1.36	24.42	20.77	0.87	56.19
	yes	nRTA	2.40	22.33	28.81	54.60	253.81	102.72	464.67
		IRTA	2.40	20.88	8.94	57.69	256.52	1.41	347.84
m_2^a	no	nRTA	0.89	2.26	5.06	2.91	5.52	13.28	29.92
		IRTA	0.89	1.92	0.43	3.00	5.68	0.66	12.59
	yes	nRTA	2.09	2.23	26.78	3.46	42.21	99.60	176.38
		IRTA	2.12	2.24	0.48	3.81	43.09	2.77	54.51
m_2^r	no	nRTA	0.92	1.61	5.13	2.93	5.50	13.28	29.36
		IRTA	0.87	2.32	0.41	2.97	5.56	0.66	12.79
	yes	nRTA	2.04	2.89	27.03	3.81	42.78	100.36	178.89
		IRTA	2.06	1.82	0.44	3.67	43.06	2.74	53.78
m_3^r	no	nRTA	1.12	82.86	5.30	179.88	30.89	13.42	313.46
		IRTA	1.10	82.44	8.12	181.21	30.55	7.63	311.06
	yes	nRTA	2.32	1351.09	28.62	2670.63	309.53	119.13	4481.32
		IRTA	2.52	1345.15	148.61	2715.98	296.11	76.70	4585.06
m_3^a	no	nRTA	1.12	66.05	5.55	181.91	31.08	13.45	299.16
		IRTA	1.07	65.95	9.72	183.97	30.53	7.58	298.82
	yes	nRTA	2.26	1318.00	35.19	2043.98	417.67	99.98	3917.08
		IRTA	2.41	1447.02	136.54	2061.64	419.39	61.59	4128.58
m_4^a	no	nRTA	1.15	3.98	5.10	18.07	13.96	13.22	55.48
		IRTA	1.15	4.03	0.46	17.94	13.15	0.79	37.52
	yes	nRTA	2.43	7.44	28.66	38.07	204.14	106.22	386.95
		IRTA	2.42	6.04	3.97	41.31	203.89	4.00	261.64
m_4^r	no	nRTA	1.17	3.16	5.54	18.13	13.32	14.53	55.84
		IRTA	1.20	3.14	0.44	19.14	13.53	0.80	38.25
	yes	nRTA	2.52	5.17	28.98	41.09	208.63	111.14	397.53
		IRTA	2.41	4.71	4.05	41.44	205.24	4.03	261.88

Table 4.4: Incremental build performance of algorithm related modifications

setup			performance units (in ms)						
mod	lib	algo	t_{ext}	t_{E1}	t_r / t_{E2}	t_{build}	t_Q	t_{reach}	$\sum$
m_5^a	no	nRTA	1.03	1.69	5.03	2.92	5.56	13.23	29.46
		IRTA	1.05	1.68	0.27	3.00	5.56	0.56	12.12
	yes	nRTA	2.09	2.07	25.61	3.53	43.92	97.60	174.82
		IRTA	2.07	2.15	0.43	3.63	44.87	2.79	55.94
m_5^r	no	nRTA	0.89	1.31	5.10	3.02	5.56	13.14	29.02
		IRTA	0.88	1.28	0.26	3.13	5.77	0.54	11.86
	yes	nRTA	2.03	1.65	25.84	3.67	43.93	98.25	175.38
		IRTA	2.03	1.65	0.41	3.60	44.54	3.06	55.29
m_6^a	no	nRTA	0.94	0.38	5.12	4.92	1.17	13.22	25.76
		IRTA	0.94	0.37	0.10	4.68	1.18	0.02	7.30
	yes	nRTA	2.16	0.47	25.91	12.94	7.46	100.60	149.54
		IRTA	2.20	0.46	0.10	12.48	7.36	0.03	22.63
m_6^r	no	nRTA	1.00	0.75	5.27	5.41	1.25	13.12	26.80
		IRTA	1.00	0.65	0.21	4.99	1.24	0.02	8.12
	yes	nRTA	2.19	0.84	26.40	13.89	7.56	102.26	153.14
		IRTA	2.11	0.81	0.18	13.33	7.79	0.02	24.24
m_7^a	no	nRTA	0.89	0.37	5.20	0.09	0.33	13.04	19.92
		IRTA	0.88	0.34	0.11	0.08	0.33	0.02	1.76
	yes	nRTA	2.14	0.37	25.85	0.09	0.97	102.50	131.91
		IRTA	2.09	0.36	0.10	0.08	0.96	0.02	3.60
m_7^r	no	nRTA	0.86	0.35	5.06	0.08	0.34	13.20	19.88
		IRTA	0.84	0.34	0.11	0.07	0.33	0.02	1.73
	yes	nRTA	2.06	0.36	25.89	0.08	0.97	101.73	131.10
		IRTA	2.02	0.35	0.10	0.07	0.95	0.02	3.52
m_8^a	no	nRTA	1.37	4.33	5.19	1.93	12.47	12.33	37.63
		IRTA	1.36	4.15	0.52	2.17	12.79	4.46	25.44
	yes	nRTA	2.52	5.88	26.39	3.05	211.75	99.67	349.26
		IRTA	2.53	5.93	4.07	2.52	213.54	61.17	289.76
m_8^r	no	nRTA	1.37	1.42	4.83	18.49	13.25	14.47	53.83
		IRTA	1.35	1.19	0.26	18.88	13.90	3.76	39.33
	yes	nRTA	2.51	1.27	25.19	38.46	215.61	99.27	382.32
		IRTA	2.52	1.22	1.24	38.77	211.37	22.91	278.03

Table 4.5: Incremental build performance of algorithm related modifications (part 2)

setup			performance units (in ms)						
mod	lib	algo	t_{ext}	t_{E1}	t_r / t_{E2}	t_{build}	t_Q	t_{reach}	$\sum$
m_9	no	nRTA	1.13	1.50	5.14	6.81	3.51	13.08	31.18
		IRTA	1.09	1.36	0.69	6.46	3.36	2.41	15.38
	yes	nRTA	2.31	1.87	28.79	16.26	32.09	107.47	188.79
		IRTA	2.40	2.75	1.59	15.68	32.65	5.31	60.38
m^a_{10}	no	nRTA	0.86	82.32	6.23	174.07	30.61	13.79	307.87
		IRTA	0.87	82.29	7.32	182.45	31.11	6.94	310.99
	yes	nRTA	2.09	1341.32	28.33	2660.96	309.02	122.79	4464.50
		IRTA	2.04	1369.43	149.18	2720.78	292.94	76.45	4610.83
m^r_{10}	no	nRTA	0.78	64.83	5.62	174.11	30.99	13.77	290.11
		IRTA	0.79	67.81	9.79	180.03	30.55	7.17	296.15
	yes	nRTA	1.92	1304.47	34.77	2092.56	425.77	106.17	3965.65
		IRTA	2.02	1461.87	127.66	2064.30	417.57	61.00	4134.44
m^s_{11}	no	nRTA	1.06	82.37	5.13	179.02	29.85	13.98	311.40
		IRTA	1.07	83.95	8.91	184.36	29.82	7.06	315.18
	yes	nRTA	2.26	1321.94	28.00	2646.02	312.52	120.54	4431.28
		IRTA	2.27	1363.30	150.16	2749.10	296.82	76.66	4638.31
m^b_{11}	no	nRTA	9.03	314.77	5.56	322.76	42.31	12.99	707.43
		IRTA	7.50	317.98	20.41	323.19	43.24	25.41	737.73
	yes	nRTA	9.29	1536.94	35.36	2685.87	392.00	113.49	4772.96
		IRTA	9.38	1573.65	171.23	2862.35	362.14	108.54	5087.29
m^s_{12}	no	nRTA	1.00	2.04	5.01	4.11	6.92	13.00	32.09
		IRTA	1.00	1.78	0.25	4.23	7.13	0.31	14.70
	yes	nRTA	2.21	2.31	30.10	5.65	71.64	108.31	220.21
		IRTA	2.27	2.19	0.45	5.23	70.80	1.04	82.00
m^b_{12}	no	nRTA	2.55	26.99	5.18	62.17	24.69	13.85	135.42
		IRTA	2.61	27.05	2.61	61.34	24.67	3.30	121.58
	yes	nRTA	4.01	47.39	31.03	116.29	280.76	111.36	590.84
		IRTA	4.08	50.31	11.49	119.09	284.30	12.21	481.48

Table 4.6: Incremental build performance of development related modifications

manually incrementalized call graph algorithm presented in this work, the update runtime is much smaller than one second for most modifications. This enables immediate analysis feedback.

4.5.4 Conclusions

As the performance figures show, the manually incrementalized implementation if fast enough to be executed alongside the incremental build, while the automatically incrementalized implementation is too slow for this purpose. The automatic incrementalization is two orders of magnitude slower than the manually incrementalized version.

The effort to develop the manually incrementalized analysis was considerably higher. The code for the Prolog based approach as presented in Section 4.3 amounts to 45 lines of code. The code for the manual approach amounts to 11,200 lines of code. The Prolog based approach took about two days to develop, whereas the Java based approach took about three months.

The modularity with respect to input depends on the kinds of changes; changes to, e.g., the start method affect the whole call graph. On the other hand, the usual changes during development occur at the leaves of the graph, far removed from start methods, thus affecting only a very small portion of the call graph.

The configuration enumerates simple patterns of source elements and can be incrementally evaluated without rebuilding an environment for automatic incrementalization.

The domain specific optimizations done for the Java based implementation decreased the runtime of this analysis so much, that it is feasible to use the analysis as part of the incremental build.

The automatically incrementalized approach is compact, elegant and has a very narrow semantic gap to the definition of the algorithm.

4.6 Related Work

A comprehensive comparison of call graph construction algorithms is presented by Grove and Chambers [69]. They define a lattice for call graph precision comparison for most of the known call graph construction algorithms. The Rapid Type Analysis is classified by its precision among other algorithms such as CHA, the context–insensitive 0–CFA, and the context–sensitive k–CFA. The latter two construct a call graph, where each call graph node is a so called contour that represents a method in a particular analysis

time. In 0–CFA each method has one contour. In k–CFA a method is analyzed in the context of k enclosing calling contours. Both 0–CFA and k–CFA create more precise call graphs than RTA, but are more costly. Currently, no incremental version for these algorithms is known.

Tip and Palsberg [130] use the RTA as a benchmark algorithm for some fix–point based call graph construction algorithms. These algorithms also provide more precise call graphs than RTA, but do not work in an incremental way.

Rountev et al. [123] integrate the RTA into the Eclipse IDE in the Tacle project[11]. There, the overall progress of a static analysis execution is approximated to provide graphical feedback with progress bars for an IDE user [121]. The RTA uses the abstract syntax tree (AST) of the source program [123]. That is an orthogonal approach to the bytecode based analysis, presented in this work. To increase the analysis performance for programs using large libraries, the libraries are analysed once and the result is stored persistently. In future runs, the result can be reused, as libraries remain unchanged. The approach discussed in this chapter either takes only library interfaces into account or analyzes the complete library dependencies in each full build. Persistent library summaries could also save time in the current approach between simulated full builds.

Souter and Pollock [126] present the first incremental call graph analysis that handles dynamically dispatched message sends. They extend Agesen's Cartesian Product Algorithm (CPA) [2] with two detailed update algorithms. An extensive runtime evaluation was not made. It is future work to evaluate how RTA and CPA compare with respect to incremental update time.

Livshits et al. [94] present an analysis that approximates reflection calls in Java. They propose a call graph construction algorithm that uses points–to information to resolve reflection calls. If reflection calls are ignored in static analysis, the results may be incomplete and unsound, because method calls and type instantiations may be missed. Reflection calls that cannot be resolved by the proposed algorithm are handled via user specifications or with an approximation that uses type cast information[12].

4.7 Chapter Summary

This chapter presented incremental call graph construction using rapid type analysis. The analysis was incrementalized and adopted to the needs of the

[11] `http://presto.cse.ohio-state.edu/tacle/index.html`

[12] This relies on the fact, that objects created by reflection usually are cast from Class objects to their specific type.

Java language. Two approaches for the implementation were discussed:

1. An automatically incrementalized approach using XSB Prolog.
2. A manually crafted incremental algorithm, implemented in Java.

Both approaches were integrated into Magellan. The evaluation of the analysis shows, that handling unrecognized control flow caused by Java specifics with simulated call sites increases the soundness of an RTA call graph. This results in a reduced number of false positives.

The automatically incrementalized approach is compact (0.5% the size of the manually incrementalized version), and has a very narrow semantic gap to the definition of the algorithm.

The modularity with respect to input depends on the kinds of changes; the usual changes during development affecting only a very small portion of the call graph. The configuration language is simple enough be incrementally evaluated without rebuilding an environment for automatic incrementalization.

The domain specific optimizations done for the Java based implementation decreased the runtime of this analysis so much, that is fast enough to be used as part of the incremental build process.

Chapter 5

Incremental Architecture Enforcement

This chapter shares some material with the paper Defining and Continuous Checking of Structural Program Dependencies [57]

Dependencies between program elements need to be modeled from different perspectives reflecting architectural, design, and implementation level decisions. To avoid erosion of the intended structure of the code, it is necessary to explicitly codify these different perspectives on the permitted dependencies and to detect violations continuously and incrementally as software evolves.

This chapter presents an approach for controlling compile time dependencies between groups of source elements. Declarative queries are used to group source elements uses into so called *ensembles*. These ensembles reach across programming language module boundaries and may overlap. The approach has been integrated into the incremental build process of Eclipse to enable continuous checking.

To show the effect of the different approaches to incrementalization, the approach was implemented twice. The first implementation is automatically incrementalized and uses XSB as the engine for tabled and incremental evaluation of logic queries. The second implementation uses a manual approach for the incrementalization of the analysis.

This chapter is structured as follows: the next section introduces the need for means to express structural dependencies between groups of source elements and details the contributions of this chapter. Section 5.2 presents means and notations for specifying these groups and constraints on their dependencies. Section 5.3 discusses the automatically incrementalized implementation and Section 5.4 the manually incrementalized implementation. Section 5.5 compares the approaches and draws conclusions. Section 5.6

presents related work and Section 5.7 summarizes the chapter.

5.1 Introduction

Constraints on structural dependencies between elements of a software need to be expressed at different levels of abstraction. For example, dependencies between layers in a layered architecture are constraints on the architectural level. An example for a design level constraint is that only factory classes can access constructors of product classes when the factory pattern [67] is employed. And stating that the fields of a certain class can only be accessed via getter and setter methods of the same class is an example for an implementation level constraint.

Expressing constraints at different levels of abstraction implies that arbitrary groups of source elements, such as class, method or field declarations, can be defined and related to each other. Each layer in a layered architecture is implemented by groups of classes including their fields and methods; the relations between these groups should be constrained in such a way, that the dependencies are, for example, only from higher layers to lower layers. The example design level constraint relates groups of classes (including their methods and fields) to groups of methods in other classes, by limiting the allowed dependencies to the product class constructor to the factory class. The example implementation level constraint relates groups of fields to groups of methods within the same class, limiting the field access to the corresponding getters and setters.

In this chapter, the term *ensemble* is used to denote logical groupings of source code elements. These ensembles are used to express structural dependency constraints. That is, classes, methods, and fields participating in the implementation of a layer may constitute an ensemble; the set of constructors of classes playing the product role in an instantiation of the factory pattern or the set of setter and getter methods of a certain class are further examples of ensembles.

Module–centric visibility mechanisms supported by programming languages, e.g., visibility modifiers of Java, are insufficient for expressing constraints on structural dependencies at different levels of abstractions, because they lack two properties, that ensembles possess:

- Ensembles can be defined orthogonal to the module system of the implementation language, e.g., the ensemble that groups all constructors of classes playing the product role in a factory pattern instantiation cuts across class boundaries. Visibility modifiers always control the visibility of a language unit; e.g. a class or a method.

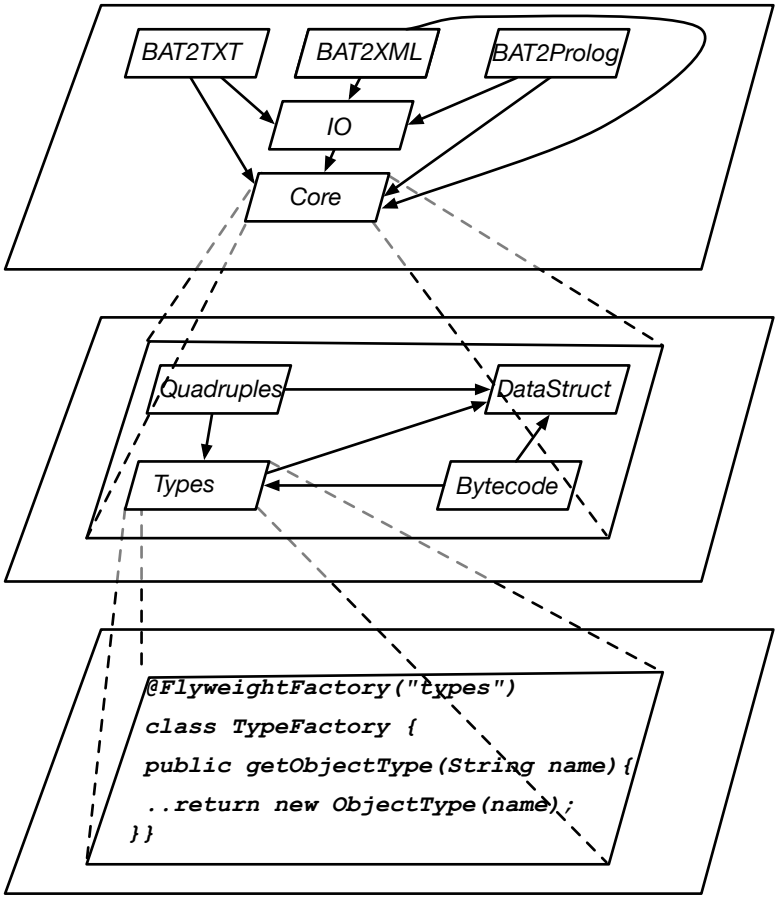


Figure 5.1: Layers of Abstraction

- Ensembles can share members. For example, a class may be part of a layer ensemble and of the product ensemble in an implementation of the factory pattern. The participation in different ensembles imposes different constraints on the allowed dependencies that are in effect simultaneously. A visualisation of this is shown in Figure 5.1, where in the top layer the major architectural building blocks of an application are shown. The middle layer zooms into one of the building blocks and shows, that the core ensemble is itself structured in ensembles. The lowest layer shows part of the code implementing ensembles. The class Typefactory shown on the lowest level is part of the `types` ensemble and part of the `core` ensemble.

The contributions of this chapter are as follows:

- An approach is presented that enables software architects and developers to express constraints on structural dependencies between arbitrary ensembles, which are continuously enforced as the software evolves. Similar to the enforcement of visibility constraints expressed by Java visibility modifiers, which happens continuously as part of the regular build process, constraints on dependencies between ensembles need to be continuously enforced as the software evolves. This ensures that the implementation of a software system always conforms to its intended dependency structure, which is crucial for code comprehension, reuse, and maintainability. Continuous checking enables developers to fix issues as soon as they occur and is a prerequisite to prevent design erosion [133], and architectural erosion [110].
- A domain–specific language embedded in Datalog [31] is proposed for defining ensembles and for expressing constraints on their dependencies. This language also supports parameterized constraint templates, which can be reused for expressing several instances of a certain constraint type. For example, in a layered architecture, one can define a template, where the implementation of any lower level is not allowed to depend on any higher level. In any concrete application, it is then sufficient to define the queries that map source elements to layers in the template. This enables the use of a predefined vocabulary for some parts of an application's architecture.
- A visual notation is proposed for the comprehensive specification of high–level architectural dependencies; its constructs are implemented in terms of the core logic–based language.
- For implementation–level specifications the use of meta–data attached to source code elements is recommended. This meta–data is used in template constraints to define dependency constraints on ensembles representing roles in design patterns.
- Checking the dependency constraints is implemented by a static analysis and integrated with the incremental build process of Eclipse. An evaluation shows that the analysis is fast enough to be used during the incremental build process of an IDE.

5.2 Specifying Dependencies

This section presents means and notations for specifying ensembles and constraints on their structural dependencies. The notations are illustrated by

constraining the structural dependencies of an example software system—the Bytecode Toolkit BAT[1], a library for analyzing Java bytecode.

5.2.1 Logic–based Core Specification Language

The domain–specific logic–based language for expressing ensembles and constraints on their dependencies, called *LogEn* (for *Log*ical *En*sembles), is embedded into Datalog[2] [31]. Datalog is a subset of Prolog that ensures termination and allows tabling of the query results without distorting the semantics of queries (for details, see Section 2.3.1 on page 44). This is a crucial prerequisite for the incrementalization of queries. Because of its good runtime efficiency compared to full Prolog, and sufficient expressiveness, Datalog is used for program queries [74]. The XSB[3]–engine is used for the implementation and therefore XSB [118] syntax and terminology is used in the following.

Source elements of Java programs are class, method and field declarations. These source elements and their relations are represented as Datalog facts. The source representation described in Section 2.3.2 is used. The following relations between Java source elements are represented: method calls, field accesses, annotations, exception declarations and exception handling constructs. The representation balances two needs: On the one hand side stands the need to conserve as much information about the structure of the Java source as is needed to enable the specification of all types of constraints related to compile time dependencies. On the other hand side stands the need to keep the representation compact enough to allow speedy conversion and to save memory. Intra–method control structures such as loops and switches are not relevant for architecture enforcement and therefore are not represented, since they neither serve as means to group source elements into ensembles nor take part in dependencies we want to control.

The uses relation between source elements is represented by `uses/2`, whose first argument is the id of a source element that uses the source element specified as the second argument. In Java, a *(syntactic) use* exists between two source elements SA and SB, if SB is used in SA. If SA and SB are method declarations, a use is e.g., a call of SB in SA. If SB is an exception type, the declaration of SA to throw SB, the creation of an instance of SB in SA, or a catch statement for exceptions of type SB are all uses of SB in SA.

```
1  part_of(E, 'Types'):-
2    type(ClassId, ClassName), prefix('bat.type.', ClassName),
3    E=ClassId|method(E, _, ClassId, _, _)|field(E, _, ClassId, _).
4
5  part_of(E, 'TypesFlyweightFactory'):-
6    type(ClassId, 'bat.type.TypeFactory'),
7    E=ClassId|method(E, _, ClassId, _, _)|field(E, _, ClassId, _).
8
9  part_of(E, 'TypesFlyweightCreation'):-
10   type(ClassId, 'bat.type.IType'),
11   inherits(SubclassId, ClassId),
12   method(E, SubclassId, '<init>', _, _).
```

Listing 5.1: Defining ensembles

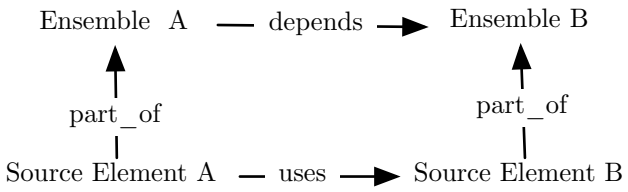


Figure 5.2: Dependencies between ensembles

Constraints on Dependencies between Ensembles

Source elements that belong to an ensemble are identified using the part_of/2 relation, where the first argument is a source element and the second parameter is the name of the ensemble. The rules defining this relation build upon the source code representation and the predefined relations described above. For illustration, three ensembles are defined in Listing 5.1. The `Types` ensemble (Line 1) comprises all source elements in any package starting with bat.type. The `TypesFlyweightFactory` ensemble matches the whole factory class (Line 5) and the `TypesFlyweightCreation` ensemble (Line 9) matches all constructors[4] of all classes that inherit from IType.

Dependencies between ensembles are derived from the uses between source elements, as schematically shown in Figure 5.2. Ensemble `A` depends on en-

[1] `http://www.st.informatik.tu-darmstadt.de/BAT`
[2] extended with arithmetic and negation using a closed world assumption
[3] `http://xsb.sf.net`
[4] methods with the name <init>

semble B, if there is a source element SA that is part of A that uses source element SB that is part of ensemble B.

To specify constraints on dependencies between ensembles violations(S, T, 'constName') is used, where 'constName' is the name of the constraint; the relation binds the source (S) and target (T) code elements of uses relations that violate constraint 'constName'. To formulate rules defining this relation, only part_of/2, conjunction, disjunction, and tabled negation [117] are used in the language. The specification as a whole is valid if violations/3 does not hold for any uses and constraints. The examples throughout the chapter and the experiments presented in Section 5.3 show that this is expressive enough to formulate a wide range of constraints.

For illustration, consider violations(S, T, 'TypesFlyweight') in Listing 5.3 on the following page. It uses the ensembles `TypesFlyweightCreation` and `TypesFlyweightFactory` defined in Listing 5.1 on the preceding page to specify that only the class TypeFactory is allowed to create instances of ObjectType and its subtypes. The rule returns all pairs `(S, T)`, such that uses(S,T) is true, `T` belongs to `TypesFlyweightCreation`, and `S` does not belong to `TypesFlyweightFactory`. For an example of a violation of the constraint expressed by violations(S, T, 'TypesFlyweight') consider the method getErasedType in Listing 5.3 on the following page and assume that it has the id m3 in the corresponding logic representation. The call in Line 10 uses the constructor of ObjectType (with the id m1 in Listing 2.7 on page 47), i.e., uses(m3, m1) is true. This violates the 'TypesFlyweight' constraint, because part_of(m1, 'TypesFlyweightCreation') is *true* and part_of(m3, 'TypesFlyweightFactory') is *false*.

Templates

In *LogEn*, templates define recurring patterns of constraints on structural dependencies. A template is a violations/3 rule, whose third argument is not a ground term (i.e., constant), but a variable or a term containing variables. Using variables in the argument representing the ensemble allows to formulate a rule that matches all ensembles that can be unified with the argument term.

For illustration, consider the template violations(S, T, flyweight(Instance)) in Listing 5.2 on the following page. This template specifies a constraint on structural dependencies between participants of the flyweight pattern, stating that only the factory is allowed to create flyweights. This specification uses the variable `Instance` in the third argument to abstract over any particular instantiation of the flyweight pattern. violations(S, T, 'TypesFlyweight') in Listing 5.3 on the next page specifies the same constraint for only one particular instantiation of the flyweight pattern, whose participants are the classes

```
violations(S,T,flyweight(Instance)) :-
    part_of(T,flyweightFactory(Instance)),
    tnot(part_of(S,flyweightCreation(Instance))).

violations(S,T,layering(Instance)) :-
    part_of(T,layer(M,Instance)),
    tnot(part_of(S,layer(N,Instance))),
    N>=M.
```

Listing 5.2: Example templates

```
violations(S, T, 'TypesFlyweight'):-
  part_of(T, 'TypesFlyweightCreation'),
  tnot(part_of(S, 'TypesFlyweightFactory')).

...

class ParameterizedType {
  ObjectType getErasedType() {
    ...
    Object erasedType = new ObjectType(fqn);
    return erasedType;
  }
}
```

Listing 5.3: A constraint and a violation of it

```
1 part_of(E, flyweightFactory('Types')):-
2   type(ClassId, 'bat.type.TypeFactory'),
3   E=ClassId|method(E, _, ClassId, _, _)|field(E, _, ClassId, _).
4
5 part_of(E, flyweightCreation('Types')):-
6   type(ClassId, 'bat.type.IType'),
7   inherits(SubclassId, ClassId),
8   method(E, SubclassId, '<init>', _, _).
```

Listing 5.4: Instantiating Templates

in the `TypeFactory` package and the class `ObjectType` and its subclasses.

To instantiate the template violations(S, T, flyweight(Instance)) for a particular instance of the flyweight pattern, the Instance variable needs to be bound to the same name for all ensembles participating in the template instantiation.

For example, Listing 5.4 shows, how to instantiate the template for the particular occurrence of the flyweight pattern in BAT. The ensembles `TypesFlyweightCreation` and `TypesFlyweightFactory` (Listing 5.1 on page 124), are changed, to replace the constants 'TypesFlyweightCreation' and 'TypesFlyweightFactory' with flyweightCreation(types) and flyweightFactory(types) respectively.

Listing 5.2 on the facing page in Lines 5 to 8 shows a template defining a constraint on structural dependencies imposed by layered architectures [26]. This template specifies that the ensemble representing the implementation of a layer N is allowed to depend on any source element in implementations of layers M. This access restriction in the layered architecture is expressed by $N >= M$; the variable Instance abstracts over any specific instance of the layered architectural pattern. To use this template in a concrete context, e.g., to specify the layered architecture of BAT, which consists of three layers the `Instance` variable needs to be bound to a concrete name, e.g., `bat`, and the extent of the resulting ensembles, `layer(0,bat)`, `layer(1,bat)`, and `layer(2,bat)`, need to be defined by specifying respective part_of/2 queries.

5.2.2 Visual Dependency Specification

This section introduces *VisEn* (for *Vis*ual *En*sembles), a visual language for specifying constraints on dependencies between high–level building blocks of an application; graphical notations facilitate the comprehension of these kinds of structural dependencies [87]. *VisEn* is mapped to *LogEn* specifications, as discussed later in this section.

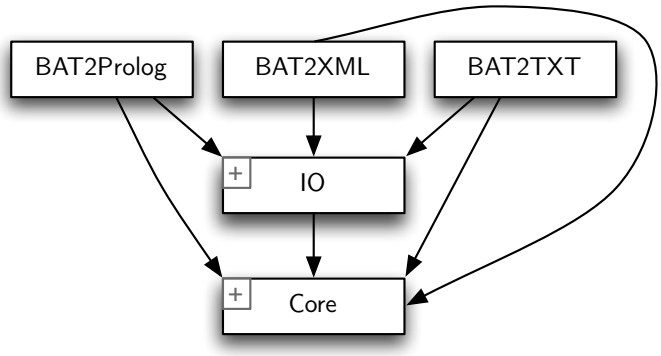


Figure 5.3: Conceptual view on BAT

Graphical Notation

VisEn is introduced by using it to express high–level constraints on structural dependencies between elements of the example system. To start with, Figure 5.3 shows a *VisEn* specification of the high–level building blocks of the BAT framework and their allowed structural dependencies. *Boxes* denote ensembles[5] and *arrows* denote allowed dependencies. The main building blocks of BAT are `Core` and `IO`; the other three ensembles in Figure 5.3 represent functionality to generate different code representations. The specification states that `Core` must not depend on any other ensemble; `Core` does not have any outgoing dependencies. `IO` may only depend on `Core`, while the other ensembles may depend on both `Core` and `IO`.

Figure 5.4 on the facing page shows a more detailed view of `Core` and `IO` and illustrates how ensembles can be nested into each other. The nesting structure has two implications on allowed dependencies. First, an element of an ensemble `E` may implicitly depend on any element that directly belongs to `E` or to any of `E`'s enclosing ensembles (e.g. source elements in the `Types` ensemble may access source elements in `Core`). All other dependencies must be stated explicitly. E.g., elements of `Writer Impl` on the left-hand side of Figure 5.4 on the next page may implicitly depend on any element `el` of `IO`, provided that `el` does not belong to a sibling sub–ensemble of `Writer Impl` such as `Reader Impl`; to enable elements of `Writer Impl` to depend on elements of `JVM Constants`, the explicit dependency from `Writer Impl` to `JVM Constants` is needed. Second, dependencies between sub–ensembles nested in two different enclosing ensembles `E1` and `E2` are allowed only if `E1` and `E2` depend on each other. This enables reasoning about ensembles in their folded state (as black boxes).

[5]The plus sign denotes ensembles with folded inner structure

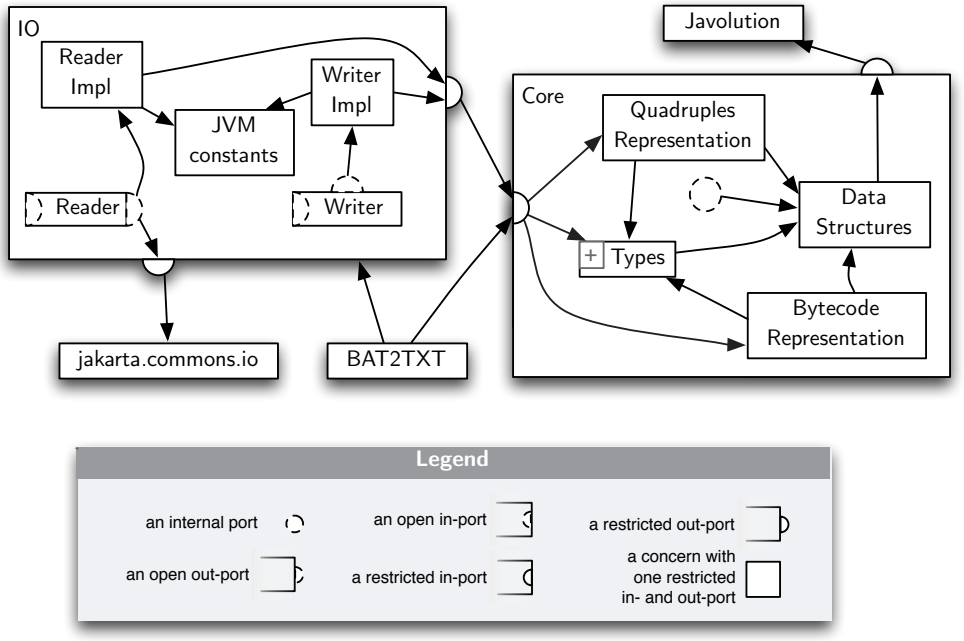


Figure 5.4: High–level architecture of (BAT)

However, an explicit dependency between `E1` and `E2` does not automatically propagate to their sub–ensembles; for this purpose so–called in– and out–ports are provided. An *out–port* is denoted by a semi–circle attached outwards to the boxes (cf. Figure 5.4) and is used to specify outgoing dependencies. An *in–port* is denoted by a semi–circle attached inwards to the boxes and collects incoming dependencies. The connection between the out–port on the right–hand side of `IO` to the in–port of `Core` specifies that `IO` as a whole depends on `Core`.

Sub–ensembles of an ensemble `E` use `E`'s out–ports to specify external dependencies, i.e., on ensembles outside `E`. The inner structure of `IO` reveals that `JVM constants` is not allowed to depend on `Core`, since it is not connected to `IO`'s corresponding out–port. An ensemble may have multiple out–ports to enable fine-granular modeling of external dependencies of its sub–ensembles. E.g., `IO` has two out–ports to express that only `Reader` may depend on the ensemble `jakarta.commons.io`, while `ReaderImpl` and `WriterImpl` may depend on `Core`. To avoid visual clutter, out–ports are only shown if there is more than one, or to specify external dependencies of a sub–ensemble; e.g., the single out–port of `Core` is shown in Figure 5.4 to specify that a dependency between `Data Structure` and `Javolution` is allowed.

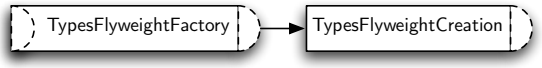


Figure 5.5: The `flyweight` pattern

An ensemble `E1` that is connected to an ensemble `E2` may access source elements that directly belong to `E2`. However, `E1` may depend only on sub–ensembles of `E2` to which `E2`'s in–port is connected. Each ensemble has exactly one in–port. If an ensemble `E` had more than one in–port, an ensemble `E1` depending on `E` would need to know about `E`'s internal structure to decide which in–port to use, making black–box reasoning impossible. In–ports are only shown if incoming dependencies need to be forwarded to nested ensembles. For illustration, consider `Core` and `IO` in Figure 5.4 on the preceding page. `Core`'s in–port has connections to sub–ensembles `Types`, `Bytecode Representation`, and `Quadruples Representation`. As a result, ensembles that depend on `Core` (`IO` and `BAT2TXT`) may depend on elements of these sub–ensembles, in addition to elements that directly belong to `Core`. `IO`'s in–port has no explicit connections, hence is not shown.

VisEn distinguishes between *open*, *restricted*, and *internal* ports. By default, a port drawn with a solid line is restricted; a hidden port is also restricted. Open ports are drawn as semi–circles with dashed lines and do not limit dependencies from or to an ensemble. For example, `Reader` and `Writer` have open in– and out–ports. `Reader` and `Writer` may access `Reader Impl` respectively `Writer Impl` and identify sets of source elements involved in reading, respectively writing, class files that are accessible to clients of `IO`. An internal port is denoted by a dashed circle and grants the elements of an ensemble access to sub–ensembles connected to it. For example, all elements in Core need to access the `Data Structures` ensemble.

Figure 5.5 shows a visual specification of the design pattern constraint that only respective flyweight factories are allowed to create flyweights of object types in BAT (cf. the 'TypesFlyweight' constraints in Listing 5.3 on page 126). This specification uses ensembles with open and restricted ports. The restricted in–port of `TypesFlyweightCreation` states that access to elements of `TypesFlyweightCreation` is exclusively granted to elements in `TypesFlyweightFactory`. Due to the open out–port, the specification does not constrain in any way dependencies of `TypesFlyweightCreation` on other source elements, e.g., helper classes.

For an example of using an internal port in a visual specification consider `Core` in Figure 5.4 on the previous page. Its inner port enables `Core`'s elements to use elements of `Data Structures`; this cannot be expressed by

```
1 e_connect(E1, E2) :-
2   (outPort(E1, P1) | internalPort(E1, P1)),
3   inPort(E2, P2),
4   p_connect_trans(P1, P2).
5
6 depends(E1, E2) :- e_connect(E1, E2) | isEnclosedIn(E1, E2).
```

Listing 5.5: Representing ensemble-dependencies

an open in–port of `Data Structures`, because this would render the latter accessible for all ensembles outside `Core` that connect to `Core`'s in–port.

From VisEn to LogEn Specifications

This section discusses, how *VisEn* notations are mapped to *LogEn*. This mapping of visual architecture specifications to Datalog is presented in the following. The Datalog facts in this section are generated from the visual specifications. *LogEn* queries use these facts to ensure the specified dependencies. The *LogEn* predicates presented in this section constitute a sub–language of *LogEn* which can be directly used to express enclosing and dependency relationships between high–level ensembles. Yet, using the visual counterpart provides a second view, which might be more convenient and might result in specifications that are easier to understand. Visual nesting of ensemble boxes in *VisEn* is transcribed to isEnclosedIn/2 facts in *LogEn*. For example, as `WriterImpl`'s box is nested into `IO`'s box in Figure 5.4 on page 129, the fact isEnclosedIn('WriterImpl', 'IO') will be generated.

The relations outPort/2, inPort/2, and internalPort/2 relate the ids of out–ports, in–ports, and internal ports respectively with their ensembles. The ids of all open ports are represented by the unary relation isOpen/1. For example, inPort('Reader', P) $\wedge$ isOpen(P) returns *true* and inPort('Javolution', P) $\wedge$ isOpen(P) returns *false*.

Ports that are directly connected in visual specifications are encoded using p_connect/2. That is, p_connect(p1, p2) holds if and only if port `p1` is directly connected with port `p2`. For example, `Writer Impl`'s out–port is directly connected with `IO`'s out–port. However, `Writer Impl`'s out–port is *not* directly connected with `Core`'s in–port. The in– and out–ports of the same ensemble are not considered connected e.g., `Types`' in- and out–ports are not connected by belonging to the same ensemble. The transitive closure of p_connect/2 is defined by p_connect_trans/2. For example, the conjunction outPort('ReaderImpl', P1) $\wedge$ inPort('Types', P2) $\wedge$ p_connect_trans(P1, P2) is *true*, since a path connecting both ports exists in Figure 5.4 on page 129.

```
1  violations(S, T, 'enclosing') :-
2   isEnclosedIn(E, EnclosingEnsemble),
3   part_of(S, E),
4   tnot(part_of(S, EnclosingEnsemble)).
5
6  violations(S, T, 'outgoing') :-
7   tnot(outPort(E, Port), isOpen(Port)),
8   (depends(E, DependentEnsemble) | E=DependentEnsemble),
9   part_of(S, E), tnot(part_of(T, DependentEnsemble)).
10
11 violations(S, T, 'incoming') :-
12  tnot(inPort(E, Port), isOpen(Port)),
13  (depends(E, DependentEnsemble) | E=DependentEnsemble),
14  tnot(part_of(S, DependentEnsemble)), part_of(T, E).
```

Listing 5.6: Queries for visually specified constraints

The rules in Listing 5.5 on the preceding page encode dependencies between ensembles. The rule depends(E1, E2) returns the ensembles on which a given ensemble depends or those depending on a given ensemble. An ensemble E depends on those ensembles with which it is connected or which encapsulate E. The rule uses e_connect(E1, E2), which holds for two ensembles that are connected in a visual specification. For example, depends('Reader Impl', E) results in E = ['IO', 'JVM Constants', 'Types', 'Quadruples Representation', 'Bytecode Representation'] and depends(E, 'Data Structures') returns E=['Core', 'Quadruples Representation', 'Bytecode Representation', 'Types'].

The violations/3 rules in Listing 5.6 define violations of the visually specified dependency relations between ensembles. The rules relate pairs of source code elements (S,T) participating in the uses/2 relation. The enclosing constraint (Line 1) states that all sources of uses relations that are elements of a sub–ensemble also belong to the enclosing ensemble. This constraint is necessary, since given two ensemble specifications $S1$ and $S2$, it is not possible to statically decide whether $S1$ matches a subset of elements matched by $S2$. The outgoing constraint (Line 6) is violated by any element of the uses(S,T) relation, where the source S is in an ensemble E that does not have an open out–port and the target T is not in E or in an ensemble on which E depends. The incoming constraint (Line 11) is violated by any element of the uses(S,T) relation, such that the target T is part of an ensemble E whose in–port is not open and the source S is part of neither E nor one of the ensembles depending on E.

5.2.3 Using Meta–Data to Define Ensembles

The primary means associating source elements and ensembles is the part_of/2 predicate of *LogEn*. Complementary to part_of/2, the approach also supports the use of meta–data as a means to associate source elements to ensembles. Definitions based on meta–data can be used for localized ensembles that participate in design and implementation level structural dependencies, as using metadata enables a definition of ensembles close to the affected source elements. For localized ensembles, such definitions tend to be less fragile and easier to comprehend and maintain when compared to logic–based specifications stored in separate artifacts. E.g., by labeling a class as a factory product, developers maintaining or using the class become aware of its role w.r.t. the factory pattern. Furthermore, if decisions about permitted design and implementation level dependencies change, it is possible to update the related specification(s) in place. The use of annotations also makes the definitions robust against refactorings, such as the renaming of affected source elements. This reduces the effort needed to maintain the ensemble structure, when compared with approaches, where the connection between the queries and the code is based on string matching. Many refactorings would break a string–based mapping, whereas the annotation–based mapping is mostly refactoring resilient, as long as the pattern that is annotated is still in place.

To illustrate the use of meta–data recall the generic specification of dependency constraints related to the flyweight pattern (cf. Listing 5.2 on page 126 in Section 5.2.1). Two generic ensembles were defined, `flyweightCreation(Instance)` and `flyweightFactory(Instance)`; the template is instantiated by binding the variable `Instance` to a name that denotes the concrete instance of the pattern, e.g., `types`. In Listing 5.1 on page 124, part_of/2 was used to associate source elements with ensembles in the dependency specification for the flyweight pattern. Using meta–data, the developer creates the `types` instance of the `flyweightFactory(Instance)` template by annotating TypeFactory with @FlyweightFactory("types") (see Lines 6–7 in Listing 5.7 on the following page).

The meta–annotation @CreateEnsemble is used to bind templates to annotations. This enables project specific annotations, as the templates can be reused and bound to different annotations. For illustration, consider Lines 1–4 in Listing 5.7 on the next page, where @CreateEnsemble defines a specific annotation for the ensemble template `flyweightFactory(X)`. The annotation @Binds binds variables of the template (here X) to parameters of the annotation. The ensemble specification then uses the annotation to get to the context of the annotation. This query is used for each instance of the generic ensemble `flyweightFactory(X)`. Using annotations makes the mapping of

```
1 @CreateEnsemble("flyweightFactory(X)")
2 @interface FlyweightFactory {
3   @Binds("X") String value;
4 }
5 ...
6 @FlyweightFactory("types")
7 class TypeFactory {...}
```

Listing 5.7: Annotations for the flyweight ensembles

source elements to ensembles explicit. Once defined, the @FlyweightFactory annotation can be used whenever a source element should be added to the respective ensemble.

In the example described in the figures above, an additional problem appears. The annotation @FlyweightFactory(x) is attached to the interface of the flyweights, but doesn't appear at the concrete types. It would be useful to provide means to ensure, that all the concrete types representing flyweights are annotated as such. For this purpose, an annotation on the interface could be used as before and an Annotation Dependency Checker (ADC, see [30]) could be employed to ensure that each class implementing a flyweight interface is marked with the appropriate annotation.

5.3 Automatic Incrementalization

This section shows, how the approach is integrated into the incremental build process of an IDE to ensure continuous enforcement of constraints

The proposed approach is implemented for Java using the static analysis platform Magellan [52, 55], which is described in detail in Section 2.2.1. Magellan is tightly integrated with the incremental build process of the Eclipse IDE and features an integration of the XSB[6] engine, which is described in Section 2.3.1. The integration into the incremental built process of Eclipse enables the implementation of queries over Java code that are continuously evaluated when code changes. Furthermore, the integration of the XSB engine supports automatic incrementalization [56, 120] of queries.

Figure 5.6 on the facing page shows an example of a development workflow using the approach.

1. The architect defines the designed structure using the visual language described in Section 5.2.2 and prepares an initial description of the

[6] `http://xsb.sourceforge.net`

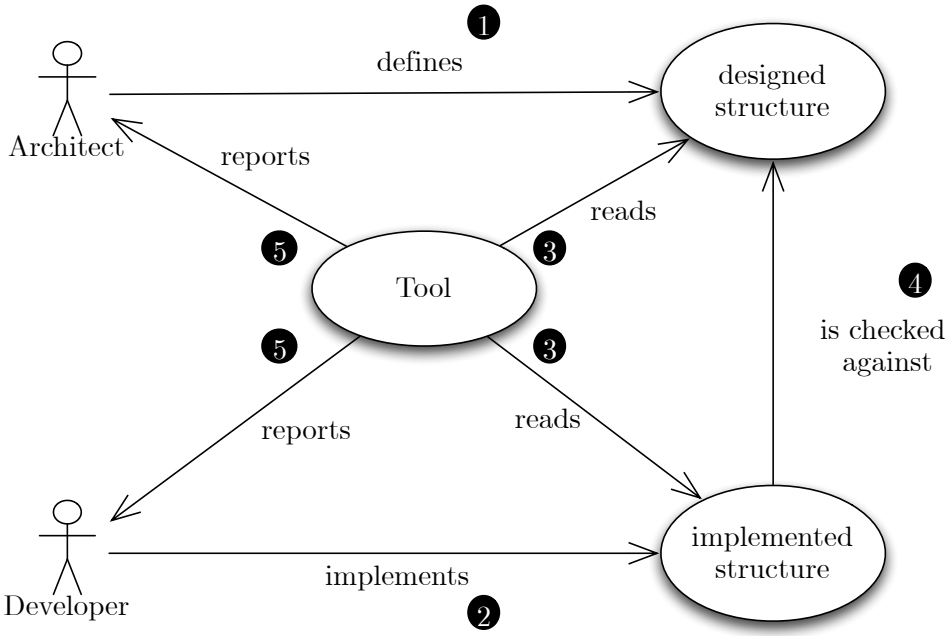


Figure 5.6: Workflow for Ensemble Based Structure Enforcement

mapping to the (not yet) implemented structure.

2. The developer then codes the implemented structure and refines the mapping. He uses annotations to instantiate templates as described in Section 5.2.3.

3. During each incremental build, the changes to the designed structure and to the implemented structure are read by the tool. All classes of a project are parsed and converted to the program model presented in Section 5.2.1 and added to the fact base. The internal representations including the mapping from designed structure to implemented structure are updated.

4. Then the implemented structure is checked against the constraints defined by the designed structure. To get the set of violations, we evaluate violations/3 (described in Section 5.2.1)

5. Any uses between source element that do not conform to these constraints are then presented as violations.

Due to the automatic incrementalization as explained in Section 2.3.1, the set of violations is incrementally maintained and the updated set of violations

is then presented to the developer.

5.4 Manual Approach to Incrementalization

The implementation of this approach is described in the Diploma Thesis of Sinisa Dukanovic [46].

This section describes an alternate implementation for the ensemble based structure enforcement. The IDE–integration and the visual specification are unchanged, compared to the automatic approach. The other elements are replaced as shown in the flowchart in Figure 5.7 on the next page: The mapping of source elements to ensembles is done with queries using XQuery [19], an XML–based query language which is described in the next section. The constraints are translated to boolean formula, which are checked using BDDs [6] (described in Section 5.4.2). The incremental changes are reported by the incremental build system of Eclipse in terms of added and removed classes. Magellan refines the changes to provide information about changes to the type hierarchy. This information is used to restrict the extent of the changes. Then the extent of the code that needs to be (re)evaluated using the BDDs is calculated with the algorithm described in Section 5.4.3.

5.4.1 XQuery

XQuery [19] is used as query language to map source elements to ensembles. XQuery is a turing complete, declarative programming language for querying XML–structured documents [23] and supports user–defined functions, external function libraries referenced by URI, and system–specific "native" functions.

Data Model

To convert the program to an XML tree, BAT2XML [48], is used, which is a library that allows a bidirectional mapping between Java bytecode and XML. The Java source elements that take part in ensembles are types, methods and fields. These elements are encoded in an XML tree with an additional program node as root node. For an example for this representation, see Listing 5.9 on page 139, which shows the XML–mapping of the Java code shown in Listing 5.8 on page 138. The declaration of the class header for DemoClass in Listing 5.8 on page 138 is shown in Line 2 of Listing 5.9 on page 139. Line 8 shows a field declaration with the return type as an inner

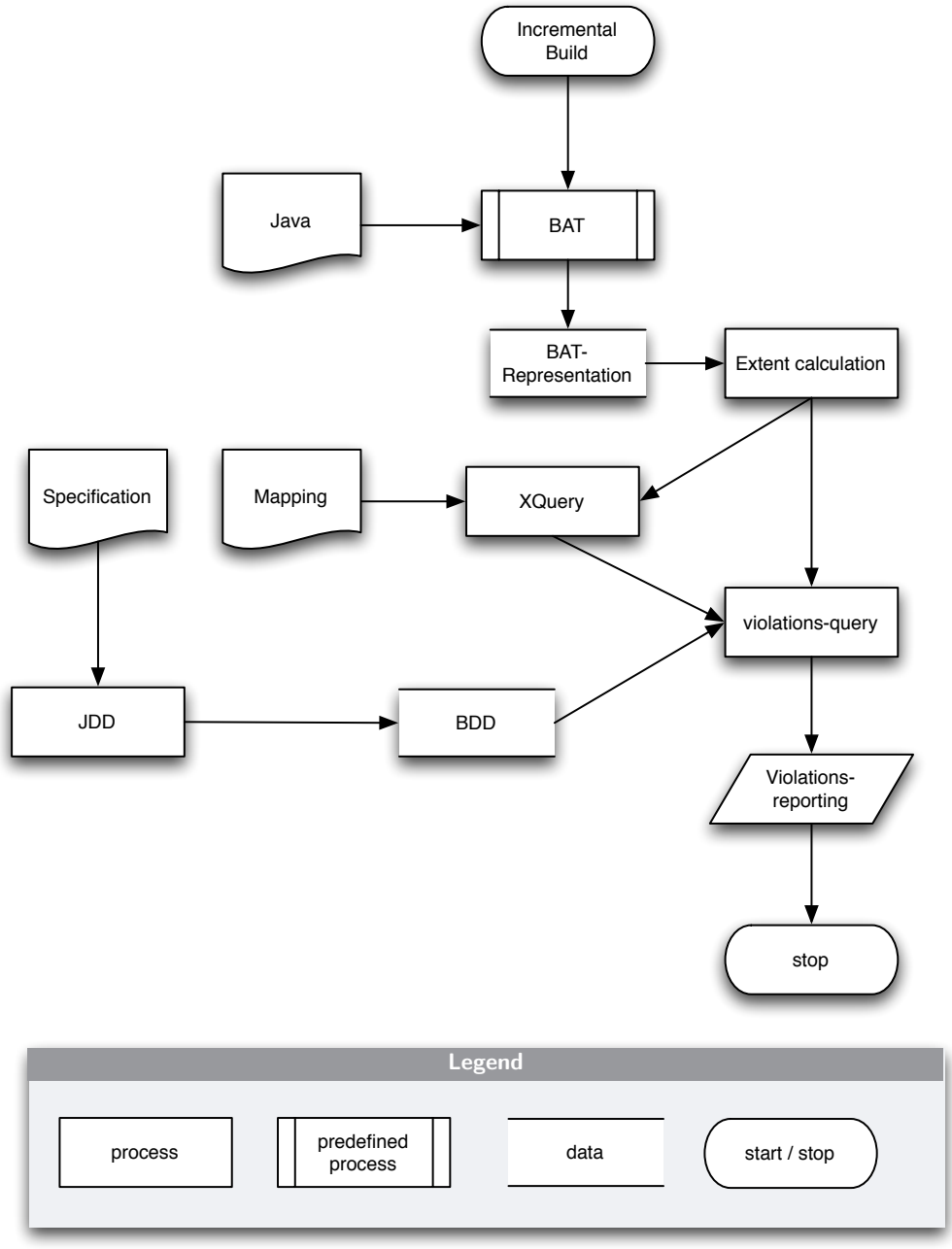


Figure 5.7: Overview of manual approach

```
1  public class DemoClass {
2    Object member;
3    public DemoClass(){
4      member = new Object();
5    }
6    public Object getMember(){
7      return member;
8    }
9    public void setMember(Object o){
10     member = o;
11   }
12 }
```

Listing 5.8: Demonstration class for the XML mapping

tag. Line 12 shows a method declaration for a constructor. Line 15 shows a getter including a return type. Line 22 shows a setter method including the encoding of a parameter.

Query Definition

To run XQueries against the XML representations, Saxon [85] is used. The most important functionality that XQuery offers for this task is the availability of so–called *path-expressions*. These expressions are defined as XPath [34] and select nodes in the XML tree. Listing 5.10 on page 140 shows an example query grouping all source elements of the class TypeFactory.

In Line 1, the namespace for the XQuery is defined. This ensures that elements with the bat namespace prefix are handled correctly by the XQuery engine. In Line 2, a path–expression is used to select all XML nodes that are children of the project node, and are nodes of the bat:class that have a property called bat:name with the value "de.tud.bat.type.TypeFactory". The class nodes are filtered by a *predicate*, a boolean expression enclosed in square brackets. The variable $class then contains all matching classes. Since fully qualified names in Java are unique, de.tud.bat.type.TypeFactory is the only class in $class. The last line of Listing 5.10 on page 140 returns the result set of the query, which contains the class de.tud.bat.type.TypeFactory including all its fields and methods.

The next XQuery, shown in Listing 5.11 on page 140 shows the use of relations between source elements to formulate queries that are more intensional than extensional. The intension of the query is to capture all initializers of IType's sub–classes. The extension of this query are the individual initialises.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <bat:class xmlns:bat="http://www.st.informatik.tu-darmstadt.de/BAT2-
       QUADRUPLES-Java1.5(7/7/05)" bat:synthetic="false" bat:name="DemoClass"
       bat:deprecated="false" bat:visibility="public" bat:abstract="false" bat:final="false
       " bat:sourcefile="DemoClass.java">
3    <bat:inherits>
4      <bat:class>
5        <bat:object-type bat:name="java.lang.Object"/>
6      </bat:class>
7    </bat:inherits>
8    <bat:field bat:name="member" bat:visibility="" bat:final="false" bat:static="false"
         bat:synthetic="false" bat:enum="false" bat:volatile="false" bat:transient="false"
9        bat:deprecated="false">
10     <bat:object-type bat:name="java.lang.Object"/>
11   </bat:field>
12   <bat:method bat:name="&amp;lt;init&amp;gt;" bat:visibility="public" bat:final="false"
         bat:static="false" bat:synthetic="false" bat:synchronized="false" bat:native="
         false" bat:strict="false" bat:deprecated="false" bat:varargs="false" bat:bridge="
         false">
13     <bat:descriptor/>
14   </bat:method>
15   <bat:method bat:name="getMember" bat:visibility="public" bat:final="false"
         bat:static="false" bat:synthetic="false" bat:synchronized="false" bat:native="
         false" bat:strict="false" bat:deprecated="false" bat:varargs="false" bat:bridge="
         false">
16     <bat:descriptor>
17       <bat:returns>
18         <bat:object-type bat:name="java.lang.Object"/>
19       </bat:returns>
20     </bat:descriptor>
21   </bat:method>
22   <bat:method bat:name="setMember" bat:visibility="public" bat:final="false"
         bat:static="false" bat:synthetic="false" bat:synchronized="false" bat:native="
         false" bat:strict="false" bat:deprecated="false" bat:varargs="false" bat:bridge="
         false">
23     <bat:descriptor>
24       <bat:parameter>
25         <bat:object-type bat:name="java.lang.Object"/>
26       </bat:parameter>
27     </bat:descriptor>
28   </bat:method>
29 </bat:class>
```

Listing 5.9: XML Representation of the demonstration class

```
1 declare namespace bat = "http://www.st.informatik.tu-darmstadt.de/BAT2
      -QUADRUPLES-Java1.5(7/7/05)";
2 let $class := ./project/bat:class[@bat:name='de.tud.bat.type.TypeFactory']
3 return $class | $class/(bat:field |bat:method)
```

Listing 5.10: XQuery for the `IType` flyweight factory

```
1 declare namespace bat = "http://www.st.informatik.tu-darmstadt.de/BAT2
      -QUADRUPLES-Java1.5(7/7/05)";
2 for $classes in ./project/bat:class where $classes/bat:inherits/bat:class/bat:
      object-type[@bat:name='de.tud.bat.type.IType']
3 let $init := $classes/bat:method[@name='\&lt;init\&gt;']
4 return $init
```

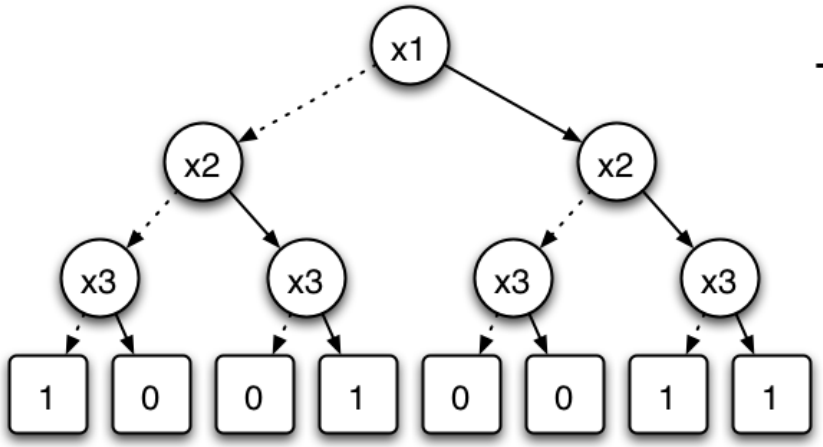
Listing 5.11: XQuery for the `IType` flyweights

But enumerating them directly would need maintenance each time the set of initialises changes in the program. By using the inheritance relationship to encode the intension directly in the query, changes to the set of initialisers are reflected in the query result without the need for manual maintenance. The query works as follows: Line 2 of Listing 5.11 assigns all sub–types of de.tud.bat.type.IType to the variable $classes. Then, in Line 3, the variable $init is assigned all methods (bat:method) that are called <init>. Since the initializers of Java classes are named <init>, all initializers are collected in $init, which is returned in the last line.

The XQueries that map the designed structure to the implemented structure of the program are read in from a file and compiled for subsequent evaluation. The queries are evaluated against an XML representation of the source code of the program under analysis. The results of the queries are sets of XML representations of classes, methods and fields. These XML representations are mapped back to the actual source elements.

5.4.2 Binary Decision Diagram

The constraints on the designed structure are written as boolean formulae, equivalent to the Datalog rules shown in Section 5.2.1 on page 123. The subset of Datalog used to describe dependency constraints is expressed in first order logic boolean expressions. The conjunction of the constraints comprises the specification. As the specification might contain a large number of constraints, and the conjunction of all constraints has to be evaluated against each use between pairs of source elements occurring in the source



x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 5.8: Example BDD

code, there is a need for optimization. For this purpose, *Binary Decision Diagrams* (BDDs, see [25]) are used. BDDs are data structures that represent boolean formulae and are widely used in formal verification, program optimization and compilers [11, 90]. To facilitate the creation and maintenance of BDDs, the Java library JDD[7] is used.

The representation of a boolean function in a BDD is a directed acyclic graph. An example is shown in Figure 5.8. The graph is comprised of decision nodes (shown as circles) and terminal nodes (shown as boxes), for *true* (represented as 1) and for *false* (represented as 0). The decision nodes represent boolean variables. The two edges leaving the node represent, if the variable is *true* (then the edge is solid) or *false* (then the edge is dotted). The two child nodes, are called high child (when connected with the solid edge) and low child (when connected with the dotted edge). This way, the evaluation of a boolean function is processed as traversal of a graph. The evaluation begins with checking the value of the boolean variable represented by the root node of the BDD. Based on the value of the root node, the next node to be queried is appointed. The procedure is repeated until a terminal node is reached.

Evaluating a boolean function in graph traversal manner, does not necessary improve the evaluation time of the boolean function. But JDD provides functionality to produce a *Reduced Ordered Binary Decision Diagram* (ROBDD, see [25]). These BDDs are called *ordered*, because the variables occur in the same order for all paths through the graph. The graph is *reduced* by repeatedly applying the following three graph transformations:

[7] `http://javaddlib.sourceforge.net/jdd/`

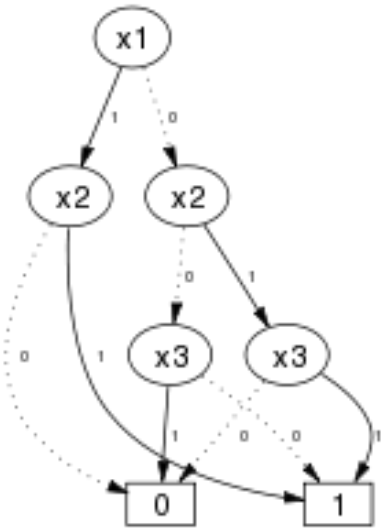


Figure 5.9: Example ROBDD

- All terminal nodes with a given label are merged into one.
- Decision nodes with identical high– and low–childs are merged into one decision node.
- If high– and low–child of a decision node are the same node, then eliminate the decision node.

An example for the result of this reduction can be seen in Figure 5.9. JDD applies the reduction and ordering algorithm to maintain a ROBDD at all times. This leads to shorter evaluation times, since the reduced BDD, has shorter paths from the root node to terminal nodes, which means that fewer boolean variables need to be evaluated.

ROBDD are used to maintain and evaluate the formula for the design specification. Since JDD applies the reduction and ordering algorithms automatically, architectural constraints are transformed to a JDD conform representation and subsequently added to a BDD (cf. Figure 5.7 on page 137). The decision nodes in the ROBDD represent $part\_of$ predicates. To evaluate, if a uses relation respects the specification, the following steps are taken. The $part\_of$ predicates in the decision nodes are evaluated with the *src* and *target* elements of the uses relation. The result determines the next node to

be processed, as described previously in this section. When a terminal node is reached, the algorithm terminates. If the terminal node is the 0-labeled node, the use is reported as violation.

5.4.3 Calculating the Extent

As discussed in Section 2.2, to calculate the subset of the program that is in need of a reanalysis, the incremental analysis needs to determine the extent of the changes to the program. To do this for the ensemble based structure enforcement, the kinds of changes have to be determined that can affect the set of violations. These are

- changes to the uses relation,
- changes to the mapping of source elements to ensembles and
- changes to the set of constraints.

Changes to parts of the source code that do not define information that is relevant for the uses relation nor the mapping relation can be ignored as they do not affect the set of violations. Examples for this kind of changes are the renaming of a local variable, or the change of a string–constant.

Algorithm 5 on the following page shows, how the extent is calculated. First, we consider the input to the algorithm. The input is comprised of the set of source elements S and their uses relation $\rightsquigarrow$, the set of ensembles E and the mapping between source elements and ensembles, denoted by the relation $\ni$. The constraints are contained in the predicate $arch\_valid$. The system after a set of changes Δ is comprises of $S', \rightsquigarrow', E', \ni', arch\_valid'$. Each individual change either adds or removes an element from one of the sets S or E, from one of the relations $\rightsquigarrow$ or $\ni$ or one of the architectural constraints from $arch\_valid$. The elements that are *added* to the system are called *new*, while the elements that are to be removed from the system are called *obsolete*. The following equation shows the relations between these sets:

$$\begin{aligned}
S' &= (S \setminus S_{obs}) &&\cup S_{new} \\
\rightsquigarrow' &= (\rightsquigarrow \setminus \rightsquigarrow_{obs}) &&\cup \rightsquigarrow_{new} \\
E' &= (E \setminus E_{obs}) &&\cup E_{new} \\
\ni' &= (\ni \setminus \ni_{obs}) &&\cup \ni_{new} \\
arch_valid' &= (arch_valid \setminus A_{obs}) &&\cup A_{new}
\end{aligned}$$

Algorithm 5 Incremental Calculation of the Set of Architecture Violations

Input:

1: S: set of source elements
2: $\rightsquigarrow$: uses relation
3: E: set of ensemble specifications
4: $\ni$: mappings between source elements and ensembles
5: $arch\_valid$: dependency constraints
6: A_{new}: new architectural constraints
7: A_{obs}: obsolete architectural constraints
8: V: set of architectural violations

Output:

9: $V = \{(src, trgt) \mid (src \rightsquigarrow' trgt) \cap \neg arch\_valid'(src, trgt)$
10: $X \leftarrow \{\}$ //uses relations that need to be checked
11: $X \leftarrow X \cup \{(s,t) \mid s \rightsquigarrow_{new} t\ \}$
12: $V \leftarrow V \setminus \{(s,t) \mid s \rightsquigarrow_{obs} t\}$
13: $X \leftarrow X \cup \{(s,t) \mid (s \rightsquigarrow t \vee t \rightsquigarrow s) \wedge (s \ni_{obs} m \vee s \ni_{new} m)\}$
14: $V \leftarrow V \cup \{(s,t) \mid (s,t) \in S', \exists a \in A_{new} \wedge \neg a(s,t)\}$
15: $V \leftarrow V \setminus \{(s,t) \mid (s,t) \in V \wedge arch\_valid'(s,t)\}$
16: $V \leftarrow V \setminus X$ // remove violations for uses we reevaluate
17: $V \leftarrow V \cup \{(s,t) \mid (s,t) \in X \wedge arch\_valid'(s,t)\}$

Further, V is the set of all violations of the architecture.

$$V = \{(src, trgt) \mid (src \rightsquigarrow trgt) \cap \neg(arch\_valid(src, trgt))\}$$

Given the system P (which is comprised of $S, \rightsquigarrow, E$ and $\ni$), the set of violations V, and a description of the changes Δ, the algorithm will update the set of violations, such that it corresponds to V'; that is, the result is the same as exhaustively checking P'. This is done by computing the set of uses that need to be reevaluated because of each change. This set is called the *check extent* of that change. The changes that affect the set of violations are:

Changes in the Uses Relation If a new use occurs, because, e.g., a call is added to a method, the new use has to be checked. If a use that is a violation is removed, the violation has to be removed. See Line 11 and 12 in Algorithm 5.

Mapping Changes If a source element is added to or removed from an ensemble, (i.e. the mapping between ensembles and source elements changes for the source element), all uses where the source element is either source or target have to be reevaluated (see Line 13 in Algorithm 5

on the facing page). As violations are only caused by the direct use of source elements, reanalyzing all direct uses is sufficient to capture all changed violations caused by changes in the ensemble mapping.

Changes to Architectural Constraints At the top level, *arch_valid* is a conjunction of architectural constraints. Thus, the formula can change in one of two ways: A constraint can be added (i.e. a formula a is changed to $a \wedge b$) or a constraint can be removed (i.e. a formula $a \wedge b$ is changed to a). All "inner" changes (like the change of a *part_of* predicate) are handled as if the term containing the old part is removed and the changed term is added.

- If a is added to arch_valid, each use of each source element in the changed program S' has to be checked against a.
  All violations in P are uses $(s \rightsquigarrow t)$ that lead to $arch\_valid(s,t)$ being false. These uses are also violations in P' (because for violations in P, $b(s,t)$ is false and false $\wedge\, a$ is always false). In addition to that, all uses in P', for which $a(s,t)$ is false are violations. To compute this set, all uses in P' have to be checked against a (see Line 14 in Algorithm 5 on the preceding page).
- If a term a is removed from $arch\_valid(s,t) = a(s,t) \wedge b(s,t)$ resulting in $arch\_valid(s,t) = b(s,t)$, all uses that are violations in P, are checked against b and removed from V', if they are no violation in P' (see Line 15 in Algorithm 5 on the facing page).

All the described extents are collected in a set X of uses that have to be rechecked, avoiding unnecessary duplication of checks. In Line 16, the uses that are to be reevaluated are removed from the set of violations, and in Line 17, the uses are checked against the changed architecture.

5.5 Comparison of the Approaches

In the following, the approach for defining a software system's structure is evaluated as well as the full build and incremental build analysis times of it. However, special emphasize is put on incremental build times as these builds are executed regularly by Eclipse whenever a document changes. Full builds are only executed on explicit user request or when a large part of a project changes, e.g. after a CVS checkout. In the latter case the user typically expects a longer build time and, hence, the additional analysis time is less critical. However, in case of incremental builds the result has to be *immediately* available as the developer directly wants to continue working.

This section compares the two approaches. The next section describes the setup of the evaluation. Section 5.5.2 presents evaluation results for the automatically incrementalized implementation. Section 5.5.3 details results for the manually incrementalized implementation. Section 5.5.4 draws conclusions from the evaluation result. Section 5.6 presents related work and Section 5.7 summarizes the chapter.

5.5.1 Setup

For the evaluation, three systems of different sizes are used to enable the reasoning about the scalability of the approach. The subject systems were:

- the Bytecode Toolkit BAT described in Section 5.2.2, consisting of 849 classes in 22 packages totaling 120,000 LOC. BAT's high–level architecture was defined as described in Section 5.2.2. The visual model of the high–level structure resulted in 17 ensembles with corresponding dependencies. The majority of these ensembles reflect the package structure, but some cut across the modular structure of Java, e.g. grouping constructors of a set of classes that create a different code representation to an ensemble. The use of annotations to model low(er) level structural dependencies resulted in another 36 ensembles.

- Jakarta regexp package[8], consisting of 14 classes with 3,663 LOC. For the Jakarta regexp package, we specified four ensembles and three constraints.

- As a large project the Aspect Bench Compiler abc [10], consisting of 2,874 classes with 285,000 LOC was used. For this project, more than 100 high level ensembles were identified.

The performance was measured on a 2.3 GHz Core2Duo system with 2 GB RAM using Eclipse 3.3 and Java 1.5. Full build as well as incremental build process time was measured. In the full build case, the whole analysis took 10 secs: 9 secs to generate the program representation (which comprises 51,278 facts) and 1 sec to check for violations.

The results of the BAT experiment are presented first; the results for the other two experiments are presented subsequently.

The annotations in BAT were used to model

- the dependencies of BAT's factories as discussed in Section 5.2.2,

[8] `http://jakarta.apache.org/regexp/`

- intra–class dependencies, e.g., that a field is only to be accessed by its getters and setters, and

- other inter–class dependencies between elements that logically belong together, but which are spread over the project for technical reasons; for example, the constructor of the class which represents a method control-flow graph is only intended to be called by the createCFG() method of the class that manages a method's code representation.

After specifying and checking for violations of the intended dependency structure, several violations were found and most of them could immediately be fixed by applying *move method*, and *move class* refactorings. However, severe violations were also found. For example, the class which manages a method's bytecode—belonging to the Core ensemble—also offered a method to clone a method's code to use it as a prototype for a new method. This method used functionality in the IO ensemble to write out the method as bytecode and then to directly reread the method. This clearly violated the intended structure where Core must not have any dependencies on IO.[9] In short, after several years of development of BAT, involving the work of a large number of (PhD) students, the main structure was still visible, but already showed structural erosion.

The changes described below were made to the BAT system to evaluate incremental analysis run times. Table 5.1 on the next page contains statistical data about the measured builds, i.e. the added and removed class files ($Class_{add}$ and $Class_{rem}$), the added respectively removed uses and the number of violations (V) found by the prototype.

- The smallest changes (Lines 2 and 3 in Table 5.1 on the following page), are the addition, respectively removal, of an empty class. A finer granulation is not provided by the Eclipse build system.

- Line 4 shows the effect of changing the inheritance hierarchy by letting the type IType inherit the interface Serializable.

- Line 5 presents the move of a class to a different ensemble. Although only a single class was moved, six classes are recompiled, because classes that reference the moved class, needed an update and, therefore, a recompilation.

[9] This particular violation could be fixed by recoding the clone functionality without resorting to IO functionality. This is left as introductory task for the next student starting work on BAT.

Nr.	Change	C_{add}	C_{rem}	added uses	removed uses	V
1	full build	1,427	0	49,114	0	119
2	add empty class	1	0	2	0	119
3	remove empty class	0	1	0	0	119
4	IType now extends Serializable	1	1	61	358	115
5	moved class to a different ensemble	6	6	357	408	123
6	change class with annotation	1	1	49	47	119
7	rename package	174	174	10,162	16,207	117

Table 5.1: Properties of code changes

- Line 6 shows the effect of editing of a class—unrelated to the system's structure—containing an annotation with a ensemble specification.
- Line 7 displays the splitting of an ensemble and moving 33 classes to a different package.

5.5.2 Automatic Incrementalization

This section discusses the results of the experiments with the automatically incrementalized implementation.

Table 5.2 on the next page lists the results of the evaluation in case of incremental builds. The type of changes performed on BAT are described in the "Change" column; the time displayed in the column "Datalog" is the overall time of creating the representation, updating the database and reevaluating the violations/3. The other columns pertain to the manually incrementalized implementation and are discussed there.

The full build analysis took 712 ms: 674 ms to build the program representation and 38 ms to check for violations. Changes comparable to the changes made for BAT were performed and in all cases the incremental analysis times were approximately $1/10^{th}$ of those in case of BAT (shown in Table 5.2 on the facing page).

For the AspectBench Compiler The full build analysis took 61.8 secs, with 60.3 secs to construct the prolog representation and 1.5 secs to actually check for violations. Again, the same type of changes were performed as in case of BAT and in this case the analysis times are 3 to 4 times slower. As

		time data in milliseconds					
	Change	Datalog	XML	Ensembles	$\rightsquigarrow_{time}$	Extent	$\sum_m$
1	full build	10,000	1,913.2	140.0	324.8	243.3	2,704.8
2	add empty class	100	775.6	140.9	0.2	0.1	916.8
3	removed empty class	156	832.1	131.3	0.1	0.1	985.1
4	IType now extends Serializable	173	832.3	136.0	1.0	0.4	981.9
5	moved class to a different ensemble	167	822.6	129.4	2.3	1.5	979.1
6	change class with annotation	122	843.4	217.4	0.5	4.6	1,064.1
7	rename package	1,285	806.1	181.8	107.6	30.1	1,210.0

Table 5.2: Performance evaluation

these results confirm, the approach does not scale linearly. Nevertheless, the performance is reasonably fast for projects with at least up to 250 KLOC.

5.5.3 Manual Incrementalization

This section examines the performance of the prototype for the manually incrementalized algorithm. The set of tests described in Section 5.5.1 is used. As subject system, the Bytecode Toolkit BAT is used.

Table 5.2 shows the timing data in milliseconds. The time needed for the generation of the XML representation of the program under analysis is given in the column titled "XML". The "Ensembles" column shows the time needed for the mapping of source elements to ensembles. The next column, titled $\rightsquigarrow_{time}$ contains the time needed to parse the class files for uses relations. The number of classes that are parsed is given in the $Class_{add}$ column, while the

number of found uses relations is shown in the column $\rightsquigarrow_{add}$. The column titled "Extent" contains the time needed to apply Algorithm 5 on page 144. The last column, titled $\sum_m$, shows the time needed for the complete set of analyses of the manually incrementalized implementation.

The full build results are given in Line 1. The time data are average values. The time measurements for each change set have been run five times and the average was taken for the respective measurement. The values in the $\sum_m$ column were computed the same way. Therefore, they may differ from the sum of XML, Ensemble, $\rightsquigarrow_{time}$ and Extent values.

The timing data from Table 5.2 on the previous page show promising results for the incremental algorithm and the parsing of the uses relations (column $\rightsquigarrow_{time}$). These two parts of the prototype are working completely incremental. The parsing for uses relations scales very well with the amount of added and deleted classes. This can be seen by comparing the minimal result in row 2 (0.2 ms), where only a single empty class had to be parsed, and the maximal result from row 7 (107.6 ms). Because of the incrementalization, this value is only dependent on the number of parsed classes and the size (lines of code) of the classes. The timing results in the extent column show that the algorithm also scales very well with the size of the change committed to a program. The size of a change is directly reflected by the number of uses that have to be analyzed.

Since the creation of the source element–ensemble mappings is not working incrementally, the values of the Ensembles column do not differ much from test to test. They remain on the same level as for a full build. Only when ensembles are added to the specification, the values increase. The values for the Ensembles column are mostly influenced by the number of ensembles, the ensemble's mapping specifications and the quality of the XQuery library.

The slowest part of the prototype is the analysis that maintains the XML Representation of the program under analysis. The timing data is provided in the column titled XML. The results in this column are either relatively stable values at about 800 ms, with slight variations depending on the amount of changed classes, or 0 ms, when no class has been altered. Although the respective analysis is implemented fully incremental, it has a shortcoming.

The analysis can be divided in two parts. The first part maintains the XML representation of the analyzed program. This part works incrementally and uses BAT's tools to generate the XML representation of the classes. Therefore, the maintenance scales well with the amount of classes that need to be added and removed. The second part prepares the XML database for the XQuery library. This task needs an average of 760 ms and therefore significantly slows down the complete evaluation time. The evaluation time ratio between the full build and the incremental builds is about 3:1 when

classes are changed and about 16:1 when only the architecture specification is changed.

The prototype uses BDDs to evaluate the architecture specification of a program. Since the BDD library performs optimizations, which "merge" the specification's constraints, it is not possible to discover the specific constraint, that causes *arch_valid* to be *false* for a uses relation. This directly affects the quality of the violation reported to the developer. Therefore, the current deviation notification only informs the developer about the source and target source elements and their respective ensemble mappings. To overcome this shortcoming, it is possible to use Java's boolean operators to evaluate the constraints. In this case, a uses relation is evaluated against each constraint individually. This way, the information about the constraint that was broken is available. This is possible without causing much overhead, if only the uses that are violations are rechecked to pinpoint the exact constraint they violate. Nevertheless this additional logic complicates the implementation.

5.5.4 Conclusions

The results indicate that the automatically incrementalized implementation for checking the constraints on structural dependencies along with the incremental build process is fast enough to be used as part of the build process. Small changes usually take less than one second, and even significant refactorings take at most two seconds. The only changes that take longer are changes of the high–level structure, which are not relevant for the day–to–day work on code.

The evaluation of the manually incrementalized implementation shows that in the current implementation, XQuery is not suitable for the mapping of source elements to ensembles, as the evaluation of the query structure takes too long. Overall, the results of the performance evaluation show that the prototype is significantly slowed down by the usage of XQuery for the mapping of source elements to ensembles. But, the other parts, i.e. the uses relations creation and validation, show very promising results and offer a solid basis for further development. The complete incremental build takes roughly about one second. The evaluation time for Magellan's complete analyses schedule for the prototype, totals at roughly 1.5 s.

The bottleneck for the manually incrementalized approach is the incremental evaluation of the ensemble specification and the maintenance of the mapping from ensembles to source elements. The configuration items are expressions over source elements and their relation (like inheritance). The used query language is intensional and not modular, as each configuration item can correspond to a set of source elements spread over the whole program.

Therefore, algorithms to evaluate incremental changes are hard to construct using manual incrementalization.

The development for the manually incrementalized version took about three months. The development of the automatically incrementalized version took about six weeks, but much of this time was spent improving the source representation and working on the XSB – Java bridge.

The code for the manually incrementalized implementation comprises 6.500 LOC and the code for the automatically incrementalized version comprises 500 LOC.

Comparing the results of the manual incrementalization with the results for the automatic incrementalization, and not considering the run time for the XQuery preparation and execution, it can be seen, that manually coding the incrementalization offers substantial performance gains. Where in the automatic incrementalization the table maintenance typically takes about 30 ms, and the actual computation of the violations another 3 ms, in the manual incrementalization, the computation of the changed uses takes up to 10 ms for the usual, smaller changes and the calculation of the violations is done in less than one millisecond in these cases.

In the automatically incrementalized implementation, Datalog is also used for the mapping between source elements and ensembles. Therefore this part is also incrementalized effectively and the interface for the developer is unified.

5.6 Related Work

Formally, the approach described above can be seen as a Relational Partition Algebra (RPA) [62, 134]. Muskens [107] uses RPA to check consistency between multiple diagrams in a UML design or between a design and an implementation, e.g. to check if a method mentioned in a UML diagram, is absent in the implementation. Crocopat [17], which translates the relational expressions to BBDBs [25] uses them to detect design patterns and structural problems in the source code, whereas the approach discussed in this chapter checks for violation of structural dependency constraints. Postma [111] describes a method for module architecture verification using RPA, which targets high–level architectural rules only.

GraphLog [37] use a graphical notation that is translated to Datalog with negation to query software structure. GraphLog focuses on the visual query language that includes graphical representations of relations, although the mapping of the visual language to code is not in their focus.

Languages specialized on software constraints like SCL [80] and its prede-

cessor FCL [81] are more expressive in the constraints they can specify when compared to the ensemble–based approach, but they lack the modularization and incrementalization features of the ensemble–based approach. SCL uses a first order predicate logic based term language to reason about program source code. The approach introduced in this chapter approach explicitly restricts the constraint language to be fast enough to evaluate and expressive enough to be useful for formulating architectural constraints. Furthermore, the authors of SCL admit, that the approach has a rather heavyweight notation. The ensemble–based approach comes with a small, developer friendly visual and metadata interface.

Rigi [105] is a reverse engineering tool that generates a layered architecture with disjunct modules. It complements the tool described in this chapter nicely, as they focus on generating a structure where the focus in this work is on maintaining and validating the structure as part of an incremental build process.

The concern manipulation environment (CME) [77] provides a unified way to represent concerns across different types of software engineering artifacts. CME includes a concern exploration tool called ConMan that represents the concern space in a tree–based view. Concerns can be assigned elements either extensionally (by explicit reference), or intensionally via a fixed set of queries over relations, such as `extends`, `implements`, `refersTo`, or `referedToBy`. The focus of CME is the modeling of concerns in the context of aspect-oriented software development; a constraint definition language is not part of the proposal. Further, the querying capabilities of CME are restricted to a predefined set of predicates.

FEAT [116] is a tool to support software maintenance, using concern graphs. Developers bind source elements to concerns and build concern graphs semi–automatically as they traverse through code. Given a concern and a source element being navigated over, e.g., a method, the developer can use a fixed set of selectors, e.g., called methods, to add structurally related elements to the same concern. Concern graphs can also be made persistent. In this approach concerns is anything a stakeholder wants to consider as conceptual unit, from design idioms to features. Concerns are structured in a tree, so each concern has a single parent concern. As concerns are scattered and tangled, source elements take part in multiple concerns. A concern graph represents the subset of a software system associated with a specific concern and consists of a collection of so called fragments, i.e. queries over the source code. The result of the query is also stored together with the fragment, to be able to compare the extensions of the fragment in different versions of the source code. Concerns overlap if they share members. Two concerns are said to *interact* with each other, if an element of one concern takes part

in a relation with the other concern. There are 23 types of relations, e.g. calls, member of, caller of, overridden by. If the source code is changed the extension of concerns change, making the fragments, and thus the concerns *inconsistent*. This can be repaired using fragment repair operators; if the extension of a concern changes, and the atoms in the query are still available, the extension is updated. if an atom is no longer available, the concern is deleted. The authors claim, that Concern Graphs facilitate the task of modifying the code implementing scattered concerns, are cost–effective and robust enough to describe concerns across different versions of a system. FEAT and our approach differ substantially in their focus. FEAT uses concerns and concern graphs for reverse engineering and comprehension purposes, to help the developer to localize and change the code implementing a certain concern. Hence, FEAT only documents existing relations between concerns, but does not impose constraints on these relations. On the contrary, ensembles are intended as a means to express and enforce constraints (invariants) over their allowed dependencies. Another difference concerns support for incremental changes. In FEAT, incremental changes of source code leads to incremental changes of the model; however, the affected relations have to be recomputed. In our approach, incremental tabling [56] is used to recompute only the part of the affected relations that depends on changed facts.

The Intensional Views [99, 100, 131] approach uses logic meta-programming to codify programming patterns ranging from best practices (getter/setter) to design patterns and bad smells. The codified patterns are used to search for instances of patterns in a code base, to check for pattern violations, or even to generate code. There are important differences between the ensemble–based approach and intensional views and their derived proposals. The focus of intentional views is very broad covering the codification of arbitrary programming patterns which are matched against existing code structures. The focus of the approach described here is on (re)grouping existing program elements into ensembles which are organized in nested dependency (usage) graphs to be continuously enforced. This difference on focus has two consequences. On the one hand, intentional views were not being used to partition program elements for the purpose of expressing dependencies across the boundaries of programming modules. On the other hand, to serve the broad focus, the program model of intentional views is much more detailed than the one used for the ensemble–based approach and their query language has unrestricted expressiveness, including the use of quantifiers and unlimited recursion. Our logic–based language is domain–specific and solely serves the specific purposes of expressing dependencies that cut across built–in modules of the programming language. The differences on expressiveness have severe effects on the suitability of the languages to be used for writing queries to be

executed along the incremental built process.

Software reflexion model [106] is an approach to prevent design erosion. The tool supports developers to check the conformance of a program to an architectural model specified by the architect. The mapping of parts of the program to disjunct modules is done declaratively by, e.g. specifying regular expressions that match the file names. The developer specifies relations ("calls" or "communicates–with") between the identified modules. The relations between the parts of the program identified as modules and relations between the modules in the specified model are compared and the result is visualized as a graph. In the ensemble–based approach the source elements can take part in more than one ensemble and can be defined at a fine level of granularity, e.g., class members can be specified to be part of a ensemble. Furthermore, the approach introduced here supports the incremental and continuous enforcement of constraints.

SonarJ[10] is a commercial tool to model the architecture of a Java program as a set of disjunct, acyclically connected modules and to check for the conformance of the program to the architecture. The granularity of SonarJ's model is fixed and on package level. A simple kind of crosscutting structure can be expressed by dividing the software in a matrix–like structure, where the rows represent technical layers like view or controller and the columns represent business layers, like supplier or customer. This structure is then resolved into disjunctive modules which are named "row::column". Dependencies and constraints are expressed using these modules. The ensemble–based approach is more flexible, as it can model ensembles that crosscut the package structure.

Like CCEL [101], our approach allows for programmer defined constraints. CCEL targets implementation restrictions like naming conventions and API usage restrictions. The approach introduced in this chapter targets the control of compile–time dependencies and unintended architectural changes.

Law Governed Architecture [103] is an approach to restrict architectural changes by enforcing laws on the architecture of evolving systems. The work includes ideas on how to anticipate certain degrees of change in the architecture of the system while maintaining the laws (comparing them to a base law, called constitution). Part of the realization of this concept is the Environment Darwin–E [104], which can check syntactic dependencies of Eiffel code using a Prolog–based static analysis. The ensemble–based approach uses a specialized, restricted language to formulate the dependencies, thus enabling fast, incremental checking.

Hedgehog [18] uses its own Prolog like language called Spine to describe declarative constraints that define implementation restrictions for de-

[10] `http://www.hello2morrow.com`

sign patterns. The constraints describe structural properties of classes and method invocations. The goal of HEDGEHOG is to verify design patterns, in the sense that developers can check that design patterns are correctly implemented. However, since HEDGEHOG tries to automatically extract the implemented design patterns it does not detect all implemented design patterns and hence is not well suited to enforce the architecture. PEC [95] is a pattern enforcing compiler for Java. Using interfaces to identify the intended design pattern, the tool combines static testing, dynamic testing (unit testing), and code generation to verify, that the pattern is implemented according to specification. Pattern–lint [122] uses a combination of static and dynamic checking to confirm that the implementation of a system maintains its expected design models and rules. These approaches all target design pattens only. The approach described in this chapter scales from modeling the top–level architecture down to intra–class design decisions.

ArchJava [4] is a Java extension, that introduces component classes. ArchJava enforces *communication integrity*. This means, that all inter–component calls follow architectural connections specified as connection patterns in the component classes. The ArchJava compiler can flag calls that violate these architectural constraints as error, or—for legacy code—as warning. Members of components are classes; the approach does not allow to model intra–class dependency constraints. While components improve the modularization, the approach does not support multiple views on the structural dependencies of the program.

Aspect–oriented languages provide modules for capturing cross–cutting concerns. The proposed approach complements aspect oriented approaches because it supports the definition of crosscutting views over programs with the goal of expressing dependency constraints between these views. The goal of AOP is to capture crosscutting behaviors in a modular way into aspects. Dependencies between code elements involved in these aspects may equally be subject of constraints.

Shomrat [125] showed, that using AspectJ [86] to enforce architectural restrictions is not an ideal choice. Although design problems are cross–cutting, they often concern static events or structural properties that cannot be captured by existing pointcut languages. Using static analysis, as is done in the prototypical implementation described above seem better suited to ensure structural properties.

Lam and Rinard present a type system and analysis for the automatic extraction of design information [89], which is used to reverse engineer design information from existing systems. The type system is an extension to Java and uses type parameters as tokens to represent design elements. The tokens are placed on objects by parameterizing the object at creation time with

the type parameter. A static analysis extracts design information from the program to construct models of the heap and object interactions. As the system does not allow the specification of architectural constraints, it is not possible to automatically detect unintended dependencies.

5.7 Chapter Summary

An approach was proposed and evaluated that integrates the specifying, visualizing and continuously checking of the structure of a software system. Central to the approach is the close integration into the incremental build process of an IDE.

At the core of the approach is a domain specific language for specifying structural dependencies between source elements. The language is designed with continuous checking in mind and enables modeling of a software system's structure at all levels of granularity; ranging from intra-class dependencies to architectural building blocks.

A visual notation directly complements the approach by facilitating the comprehensive modeling of a system's high level structure. By using annotations (meta–data), a refactoring resilient modeling of low(er) level structural dependencies was enabled.

Two implementations of the approach were discussed:

- an automatically incrementalized implementation based on Datalog, running in XSB
- a manually incrementalized implementation based on XQuery and BDDs.

The automatically incrementalized implementation proved to be fast enough to be used as part of the incremental build process. The manual approach showed partly promising results, although the chosen XQuery engine was too slow to allow the use of the implementation in incremental builds.

Chapter 6

Conclusions and Future Work

This chapter presents the conclusions drawn from this work and gives an outlook to possible future work. The following section summarizes the work up to this point and discusses the conclusions. Section 6.2 presents the outlook.

6.1 Conclusions

The goal of this thesis was to ease the integration of static analyses into IDEs, while retaining the advantages of the incremental build process. Static analyses get more and more integrated into today's IDEs. But these analyses are not integrated into the incremental build process and therefore lack immediate feedback and integration into the development workflow. Also, the analysis platforms are closed for third party analyses and do not allow reuse of existing analyses and their results.

An open platform offering means for incrementalization of analyses was lacking. Also, criteria to compare and select incrementalization approaches were needed to allow analysis developers to make an informed choice with respect to the appropriate approach.

In this thesis, two different approaches to the incrementalization of static analyses when integrating them into IDEs were described. The approaches were evaluated and compared to each other.

In Chapter 2, the approaches to achieve incrementalization of static analyses were categorized into automatic and manual incrementalization. Platforms were presented, that support the development of static analyses using an approach of either category.

For this thesis, three static analyses were selected, that represent different

categories of analyses.

- As an example of static analyses that detect data flow properties, incremental implementations of the confined types analysis were discussed in Chapter 3.

- As an example for static analyses that check for control flow properties, incremental implementations of the rapid type analysis were presented in Chapter 4. The rapid analysis creates and incrementally maintains an interprocedural call graph.

- An example for static analyses that check for structural properties was discussed in Chapter 5. The ensemble–based architecture enforcement analysis incrementally checks for violations of constrains on dependencies between groups of source elements.

The analyses were examined for the expressiveness of their configuration language and for their modularity with respect to input and data structures.

Each analysis was implemented twice; once using automatic incrementalization and once using manual incrementalization. The implementations were compared according to development time, size of code and runtime of the analyses.

The following conclusions were drawn from the implementations of the static analyses described in the Chapters 3 to 5:

- The implementations of the confined types analyses, as described in Chapter 3 are in both cases fast enough to enable the use of confined types in day-to-day development, although the manually incrementalized implementation is about 40 times faster. The automatically incrementalized implementation was developed ten times faster and is ten times smaller than the manually incrementalized implementation. The analysis is modular with a class-based scope, using the inheritance hierarchy as the only whole-program fact. The extent of program changes is relative small and easy to compute. The confined types analysis is configured using metadata attached to source elements. In Java this is realized by attaching annotations to types and methods. This form of configuration languages is extensional and modular, as each configuration item corresponds to exactly one source element.

- The incremental rapid type analysis (RTA) as discussed in Chapter 4 is fast enough for use in the incremental build process, if implemented using the manual approach (about 150 ms for small changes). The implementation using the automatic approach is two orders of magnitude

slower, but was developed in two days and only amounts to 45 LOC. The analysis is configured using regular expressions over source elements to, e.g., signify start methods. The language is sufficiently small, as to be efficiently evaluated incrementally using a general purpose programming language. The interprocedural call graph itself is an whole-program data structure, but—using domain knowledge—this data structure can be stored and updated efficiently using fast, random-access data structures like hash maps. The automatically incrementalized implementation is compact, elegant and has a very narrow semantic gap to the definition of the algorithm.

- For the ensemble based architecture enforcement (see Chapter 5), the implementation using automatic incrementalization is fast enough, while the manually incrementalized implementation is an order of magnitude slower. The bottleneck for the manually incrementalized approach is the incremental evaluation of the ensemble specification and the maintenance of the mapping from ensembles to source elements. These mappings are whole-program artifacts generated from a specification language that requires the expressiveness of Datalog.

  The configuration items are expressions over source elements and their relation (like inheritance). The used query language is intensional and not modular, as each configuration item can correspond to a set of source elements spread over the whole program. Therefore, algorithms to evaluate incremental changes are hard to construct using manual incrementalization. Incrementally incorporating changes to configurations written in such a language requires an incrementalization environment that is close to a general purpose incrementalization environment for a programming language.

  The effort necessary to develop a specialized environment for the configuration language is close to the effort needed to develop a general purpose incrementalization environment. In cases like this, it is advisable to use (or adapt) an existing general purpose environment that provides automatic incrementalization.

These conclusions can be generalized to the following comparison of the approaches to incrementalization:

- Analyses that are modular with respect to their input can successfully be incrementalized using the manual approach. As an example, see the confined type analysis. But as shown by the incremental rapid

type analysis, the reverse is not always true. The IRTA is not modular with respect to its input, but domain knowledge was exploited to achieve a runtime advantage for the manually incrementalized implementation. When domain knowledge can be used to remove the need for a general purpose incrementalization environment, the runtime advantages of manually crafted algorithms can be exploited and the manually incrementalized approach has clear advantages in terms of runtime performance.

On the other hand, the environment that supports automatic incrementalization uses a declarative language, namely Datalog. Thus, analyses can be specified in a purely declarative way, which leads to representations with a smaller representational gap, when compared to imperative environments like Java. As a declarative language, Datalog supports a programming model where the developer can concentrate on specifying the problem and the runtime environment searches for the solutions. When implementing the analyses in an imperative language like Java, developers can (and must) not only specify the problem, but also define the algorithms and supporting data structures to search for the solutions.

XSB, the logic programming system used for the automatic incrementalization also supports Prolog, which is more expressive than Datalog. Nevertheless, Datalog proved to be a better fit for the analyses examined in this thesis:

  - Prolog is not purely declarative and exposes specifics of its runtime implementation making optimisations possible (and sometimes necessary; Knuutila [88] stated, that "careless declarative programming [in Prolog] usually leads to programs that are very slow—or impossible—to execute on current computers". Datalog is much closer to being a purely declarative language, as it does not contain the non-declarative constructs of Prolog (e.g. cuts).
  - Using tabling further simplifies the programming model, as tabling softens the performance impact of statement ordering (see Section 2.3.1).

- The work spent on improving implementations had effects of different scopes: Effort spent on speeding up parts of the automatically incrementalized version by refining the platform (e.g., improving the Datalog code representation or improving the XSB-Runtime environment) benefits all analyses that build on this platform. When following the

manual incrementalization approach, the effort spent on improving the manually incrementalized algorithm only benefits this implementation.

This distinction favors the automatic incrementalization approach, because it enables easier reuse. The work invested in improving the environment for one analyses pays off when developing other analyses.

6.2 Future Work

This section discusses an outlook on possible future work.

The configuration languages of the implemented analyses have different expressiveness. The extensional markups of the confined types analysis and the regular expressions of the incremental RTA were easy to maintain using the Java-based approach. The queries of the ensemble based architecture enforcement are on the expressive level of Datalog. Changes to this configuration language are easier to maintain using an environment with automatic incrementalization like XSB. What remains to be explored is the level of expressive power for configuration language that requires automatic incrementalization.

Another direction for future work aims to close the gap between the Java runtime environment and the runtime environment for automatic incrementalization, as it proved to be a burden from multiple perspectives:

- The semantic gap between the environments is quite large. Switching between the object oriented, imperative mindset and the declarative, specification-driven mindset of Datalog programming means switching between different abstractions.

- The transaction cost in terms of runtime performance for moving between the environments is quite high, despite major improvements during the course of this thesis.

The LINQ [98] project provides some interesting ideas, that could lead to means to lessen the semantic gap between the Java runtime environment and the runtime environment for automatic incrementalization. LINQ defines a design pattern of general purpose standard query operators for traversal, filter, and projection of arbitrary data sources. Based on this pattern, any .NET language can define special query comprehension syntax that is subsequently compiled into these standard operators. Applying this for the development of incremental static analysis would simplify the querying stage of the analysis.

To reduce the transaction costs for moving between the runtime environments, it seems promising to extend tuProlog [44] with incremental tabling. tuProlog is a Java-based, light–weight Prolog engine. Developers create components of a tuProlog application by choosing at any step the most suitable paradigm, and use either Prolog for declarative implementation or Java for imperative parts [45].

In the current implementations following the approach for automatic incrementalization, the number of transitions between the runtime environments is minimized, because the transaction costs still are substantial (in the order of 100 milliseconds). The preparation of the source representation is done in the Java environment, because there are fast algorithms in place that can be reused. Then, the transition to the XSB runtime is made, where the incremental algorithm is computed. For the presentation of feedback to developers, the transition back to the Java environment is made.

If the transaction cost of moving between the environments is near zero, it becomes possible to intermix the two described approaches. Parts of the static analysis are incrementalized using specially crafted algorithms while other parts are incrementalized using automatic incrementalization. Mixing approaches allows to combine the speed improvements from the manual approach with the simple programming model of the automatic approach.

Scientific Career

04/2003-11/2008
Darmstadt University of Technology
PhD student in the Software Technology Group of Prof. Mira Mezini

10/1992-09/1999
Darmstadt University of Technology
Studies in Computer Science. Graduated as Diplom–Informatiker

Bibliography

[1] Michael Achenbach. Incremental rapid type analysis for full java. Master's thesis, TU Darmstadt, 2007.

[2] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) '95*, pages 2–26, London, UK, 1995. Springer-Verlag. ISBN 3540601600. URL `http://portal.acm.org/citation.cfm?id=679533`.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1986.

[4] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *Proceedings of the international conference on Software engineering (ICSE) 2002*. ACM Press, 2002. ISBN 158113472X. doi: 10.1145/581339.581365. URL `http://dx.doi.org/10.1145/581339.581365`.

[5] José Júlio Alferes, Carlos Viegas Damasio, and Luís Moniz Pereira. SLX-A top-down derivation procedure for programs with explicit negation. *International Logic Programming Symp*, pages 424–439, 1994.

[6] Henrik Reif Andersen. An introduction to binary decision diagrams, 10 1997. URL `http://www.itu.dk/~hra/notes-index.html`.

[7] Andrew W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, 1998.

[8] Cyrille Artho and Konstantin Knizhnik. Jlint. `http://artho.com/jlint/`, June 2004.

[9] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2002.

[10] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *Proceedings of the international conference on aspect-oriented software development (AOSD) 2005*, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-042-6. doi: http://doi.acm.org/10.1145/1052898.1052906.

[11] A. Aziz, F. Balarin, S.T. Cheng, R. Hojati, T. Kam, SC Krishnan, RK Ranjan, TR Shiple, V. Singhal, S. Tasiran, et al. HSIS: a BDD-based environment for formal verification. *Proceedings of the 31st annual conference on Design automation*, pages 454–459, 1994.

[12] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. ACM Press. ISBN 0-89791-788-X. doi: http://doi.acm.org/10.1145/236337.236371.

[13] David Francis Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California, Berkeley, 1997. URL `http://researchweb.watson.ibm.com/people/d/dfb/thesis.html`.

[14] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL) 2002*. ACM Press, 2002.

[15] Robert Balzer, Jens Jahnke, Marin Litoiu, Hausi A. Müller, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Ken Wong. 3rd international workshop on adoption-centric software engineering. In *Proceedings of the international conference on Software engineering (ICSE) 2003*, 2003.

[16] BCEL. Bcel, 2005. URL `http://jakarta.apache.org/bcel/`.

[17] Dirk Beyer, Claus Lewerentz, and Andreas Noack. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31, 2005. URL `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1401929`.

[18] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of design patterns in java. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE). 2005*, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-993-4. doi: http://doi.acm.org/10.1145/1101908.1101943.

[19] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. Xquery 1.0: An xml query language. W3c working draft 04 april 2005, W3C, 2005. `http://www.w3.org/TR/xquery/`.

[20] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001. ISSN 0018-9162.

[21] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, November 2002. ACM Press. ISBN 1581134711. doi: 10.1145/582419.582440. URL `http://dx.doi.org/10.1145/582419.582440`.

[22] Gilad Bracha. Pluggable type systems. In *Workshop on Revival of Dynamic Languages as part of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA) 2004*, 2004.

[23] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0. *W3C Recommendation*, 6, 2000.

[24] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering,. *Computer*, 20(4):10–19, April 1987. URL `http://portal.acm.org/citation.cfm?id=26441`.

[25] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3), 1992. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/136035.136043.

[26] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., 1996. ISBN 0-471-95869-7.

[27] Miguel Calejo. InterProlog, a declarative Java-Prolog interface. *Procs. Logic Programming for Artificial Intelligence and Information Systems (thematic Workshop of the 10th Portuguese Conference on Artificial Intelligence), Porto, December*, 2001.

[28] Miguel Calejo. InterProlog: Towards a declarative embedding of logic programming in Java. *Theory and Practice of Logic Programming, Lecture Notes in Computer Science*, 3229:714–717, 2004.

[29] Steven Carroll and Constantine Polychronopoulos. A framework for incremental extensible compiler construction. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 53–62, New York, NY, USA, 2003. ACM. ISBN 1-58113-733-8.

[30] Vasian Cepa and Mira Mezini. Declaring and Enforcing Dependencies Between .NET Custom Attributes. *In Procedings of the Third International Conference on Generative Programming and Component Engineering (GPCE'04) - LNCS 3286*, pages 283–297, 2004. URL `http://www.springerlink.com/index/A59VJ1APP4XE3QC9`.

[31] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1), 1989.

[32] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM (JACM)*, 43(1):20–74, 1996.

[33] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. *SIGPLAN Not.*, 34(10):1–19, 1999. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/320385.320386.

[34] James Clark and Steve DeRose. Xml path language (xpath) version 1.0. W3c recommendation 16 november 1999, W3C, 2005. `http://www.w3.org/TR/xpath/`.

[35] Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: deployment-time confinement checking. In *Proceedings of the annual ACM SIGPLAN conference on Object-oriented pro-*

gramming systems, languages, and applications (OOPSLA) 2003. ACM Press, 2003.

[36] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA) 1998*, New York, NY, USA, 1998. ACM Press. ISBN 1-58113-005-8. doi: http://doi.acm.org/10.1145/286936.286947.

[37] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *Proceedings of the international conference on Software engineering (ICSE) '92*, pages 138–156, New York, NY, USA, 1992. ACM Press. ISBN 0-89791-504-6. doi: http://doi.acm.org/10.1145/143062.143106.

[38] Tom Copeland, et. al. Pmd 3.0. `http://pmd.sourceforge.net/`, March 2005.

[39] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001. ISBN 0-262-03293-7.

[40] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952:77–101, 1995. URL `http://citeseer.ist.psu.edu/156383.html`.

[41] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 105–116, New York, NY, USA, 1981. ACM. ISBN 0-89791-029-X.

[42] Linda G. DeMichiel. *Enterprise JavaBeans Specification, Version 2.1*. SUN Microsystems, 2003.

[43] Bart Demoen and Konstantinos Sagonas. Memory management for prolog with tabling. In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 97–106, New York, NY, USA, 1998. ACM. ISBN 1-58113-114-3.

[44] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In

I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 184–198. Springer, 2001. ISBN 978-3-540-41768-2. 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings.

[45] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Science of Computer Programming*, 57(2):217–250, August 2005. ISSN 0167-6423.

[46] Sinisa Dukanovic. Encoding and incremental checking of application architectures. Master's thesis, Technische Universität Darmstadt, 2007.

[47] Eclipse. Eclipse 3.1, 2005. URL `http://www.eclipse.org`.

[48] Michael Eichberg. Bat2xml: Xml-based java bytecode representation. *Electronic Notes in Theoretical Computer Science*, 141(1):93–107, December 2005. Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2005).

[49] Michael Eichberg. *Open Integrated Development and Analysis Environments*. PhD thesis, TU Darmstadt, Fachbereich Informatik, 2007.

[50] Michael Eichberg and Christoph Bockisch. Bat, 2005. URL `http://www.st.informatik.tu-darmstadt.de/BAT`.

[51] Michael Eichberg and Christoph Bockisch. Magellan, 2005. URL `http://www.st.informatik.tu-darmstadt.de/Magellan`.

[52] Michael Eichberg, Mira Mezini, Klaus Ostermann, and Thorsten Schäfer. Xirc: A kernel for cross-artifact information engineering in software development environments. In *Proceedings of WCRE'04*. IEEE Computer Society, 2004.

[53] Michael Eichberg, Mira Mezini, Thorsten Schäfer, Claus Beringer, and Karl-Matthias Hamel. Enforcing system-wide properties. In *Proceedings of ASWEC 2004*. IEEE Computer Society, 2004.

[54] Michael Eichberg, Sven Kloppenburg, Mira Mezini, and Tobias Schuh. Incremental confined types analysis. In *Proceedings of the Sixth Workshop on Language Descriptions, Tools and Applications (LDTA)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.

[55] Michael Eichberg, Mira Mezini, Sven Kloppenburg, Klaus Ostermann, and Benjamin Rank. Integrating and scheduling an open set of static

analyses. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE) 2006*. IEEE Computer Society, 2006.

[56] Michael Eichberg, Matthias Kahl, Diptikalyan Saha, Mira Mezini, and Klaus Ostermann. Automatic incrementalization of Prolog based static analyses. In *Proceedings of PADL'07*. Springer, 2007.

[57] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. *Proceedings of the international conference on Software engineering (ICSE) 2008*, 2008.

[58] Rudolf Eisler. *Wörterbuch der philosophischen Begriffe*. ES Mittler und sohn, 1904.

[59] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM Press. ISBN 1581137575. URL `http://dx.doi.org/10.1145/945445.945468`.

[60] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of OSDI 2000*. Usenix, 2000.

[61] Jean-Marie Favre, Jacky Estublier, and Remy Sanlaville. 3rd international workshop on adoption-centric software engineering. In *Proceedings of the international conference on Software engineering (ICSE) 2003*, 2003.

[62] Loe M. G. Feijs and Rob C. van Ommering. Relation partition algebra – mathematical aspects of uses and part-of relations. *Science of Computer Programming*, 33(2), 1999. URL `http://www.sciencedirect.com/science/article/B6V17-3VX92V4-3/2/1f79a6128c152c0abf39ad0bf2274147`.

[63] Stuart Feldman. Make - a program for maintaining computer programs. *Software-Practice and Experience*, pages 255–265, April 1979. URL `http://wolfram.schneider.org/bsd/7thEdManVol2/make/make.html`.

[64] Matthias Felleisen. On the expressive power of programming languages. In *ESOP '90: Proceedings of the 3rd European Symposium on Programming*, pages 134–151, London, UK, 1990. Springer-Verlag. ISBN 3-540-52592-0.

[65] Philip W. L. Fong. Link-time enforcement of confined types for jvm bytecode. In *Proceedings of the Annual Conference on Privacy, Security and Trust (PST) 2005*, October 2005.

[66] Philip W. L. Fong. Reasoning about safety properties in a jvm-like environment. *Sci. Comput. Program.*, 67(2-3):278–300, 2007. ISSN 0167-6423.

[67] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995. ISBN 0201633612. URL `http://www.amazon.co.uk/exec/obidos/ASIN/0201633612`.

[68] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Proceedings of the annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA) 2001*. ACM Press, 2001.

[69] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions Program. Lang. Syst.*, 23 (6):685–746, November 2001. ISSN 0164-0925. doi: 10.1145/506315.506316. URL `http://portal.acm.org/citation.cfm?id=506316`.

[70] David Grove, Greg Defouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 32, pages 108–124, New York, NY, USA, October 1997. ACM Press. doi: 10.1145/263698.264352. URL `http://dx.doi.org/10.1145/263698.264352`.

[71] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications, 1995.

[72] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. Maintaining views incrementally. In *Proceedings of SIGMOD international conference on management of data*. ACM Press, 1993.

[73] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris de Volder. Codequest: querying source code with datalog. In *Proceeding of the annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA) 2005*, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-193-7. doi: http://doi.acm.org/10.1145/1094855.1094884.

[74] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)'06*, 2006. doi: 10.1007/11785477_2. URL `http://dx.doi.org/10.1007/11785477_2`.

[75] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI). 2002*. ACM Press, 2002.

[76] Elliotte Rusty Harold. Zap bugs with pmd. `http://www-106.ibm.com/developerworks/java/library/j-pmd/`, January 2005.

[77] William Harrison, Harold Ossher, Stanley Sutton, and Peri Tarr. Concern modeling in the concern manipulation environment. In *: Proceedings of MACS '05*, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-119-8. doi: http://doi.acm.org/10.1145/1083125.1083134.

[78] John Hogg. Islands: aliasing protection in object-oriented languages. In *Proceedings of the annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA) 1991*, New York, NY, USA, 1991. ACM Press. ISBN 0-201-55417-8. doi: http://doi.acm.org/10.1145/117954.117975.

[79] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992. ISSN 1055-6400. doi: http://doi.acm.org/10.1145/130943.130947.

[80] Daqing Hou and H. James Hoover. Using SCL to specify and check design intent in source code. *IEEE TSE*, 2006. URL `http://www.cs.ualberta.ca/~daqing/projects/SCL.html`.

[81] Daqing Hou, James H. Hoover, and Piotr Rudnicki. Specifying framework constraints with FCL. In *Proceedings of CASCON'04*. IBM Press, 2004. URL `http://portal.acm.org/citation.cfm?id=1034922`.

[82] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12), 2004.

[83] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996. ISBN 0-471-94148-4.

[84] Matthias Kahl. Automatic incrementalisation of prolog-based static analyses. Master's thesis, Technische Universität Darmstadt, 2006.

[85] Michael Kay. Saxon 8.4. `http://saxon.sourceforge.net`, 2005.

[86] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)'01*. Springer, 2001.

[87] Jens Knodel, Dirk Muthig, and Matthias Naab. Understanding software architectures by visualization–an experiment with graphical elements. In *Proceedings of WCRE '06*, 2006. ISBN 1095-1350.

[88] Timo Knuutila. Efficient prolog programming efficient prolog programming efficient prolog programming. *SOFTWARE—PRACTICE AND EXPERIENCE*, 22(3):209–221, 3 1992.

[89] Patrick Lam and Martin Rinard. A type system and analysis for the automatic extraction and enforcement of design information. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)'2003*, 2003. doi: http://dx.doi.org/10.1007/b11832. URL `citeseer.ist.psu.edu/article/lam03type.html`.

[90] Rainer Leupers and Peter Marwedel. A BDD-based frontend for retargetable compilers. *Proceedings of the 1995 European conference on Design and Test*, 1995.

[91] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for java. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 73–79, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-413-4. doi: http://doi.acm.org/10.1145/379605.379676.

[92] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[93] Benjamin Livshits. Findings security errors in java applications using lightweight static analysis. In Annual Computer Security Applications Conference, Work-in-Progress Report, November 2004.

[94] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Programming Languages and Systems: Third Asian Symposium, APLAS 2005*, volume 3780, pages 139–160, Berlin, November 2005. Springer-Verlag. doi: 10.1007/11575467_11. URL `http://dx.doi.org/10.1007/11575467_11`.

[95] Howard C. Lovatt, Anthony M. Sloane, and Dominic R. Verity. A pattern enforcing compiler (PEC) for Java: using the compiler. In *Proceedings of APCCM'05*. Australian Computer Society, Inc., 2005. ISBN 1-920-68225-2.

[96] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094840.

[97] Enric Mayol and Ernest Teniente. A survey of current methods for integrity constraint maintenance and view updating, 1999. URL `citeseer.ifi.unizh.ch/mayol99survey.html`.

[98] Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, 2006.

[99] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Expert Systems with Applications*, 23(4), 2002. URL `http://www.sciencedirect.com/science/article/B6V03-46W0J3J-9/2/983734a0a3bdce78f3433a9b8af8fc34`.

[100] Kim Mens, Andy Kellens, Frederic Pluquet, and Roel Wuyts. Co-evolving code and design with intensional views: A case study. *Computer Languages, Systems & Structures*, 32(2-3),

2006. URL http://www.sciencedirect.com/science/article/B73H1-4HJ47XW-1/2/dd4866158d0ddaf6ef9cc10a342b29f1.

[101] Scott Meyers, Carolyn K. Duby, and Steven P. Reiss. Constraining the structure and style of object-oriented programs. In *Principles and Practice of Constraint Programming*, 1993. URL http://citeseer.ist.psu.edu/meyers93constraining.html.

[102] Microsoft. Prefast with driver-specific rules. http://www.microsoft.com/whdc/devtools/tools/PREfast-drv.mspx, October 2004.

[103] Naftaly H. Minsky. Should architectural principles be enforced? *Computer Security, Dependability and Assurance: From Needs to Solutions*, 1998. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=798359.

[104] Naftaly H. Minsky and Partha Pratim Pal. Law-governed regularities in object systems. part 2: a concrete implementation. *Theor. Pract. Object Syst.*, 3(2), 1997. ISSN 1074-3227. doi: http://dx.doi.org/10.1002/(SICI)1096-9942(1997)3:2<87::AID-TAPO2>3.3.CO;2-P.

[105] Hausi A. Muller, M.A. Orgun, S.R. Tilley, and J.S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

[106] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of FSE'95*. ACM Press, 1995. ISBN 0897917162. doi: 10.1145/222124.222136. URL http://dx.doi.org/10.1145/222124.222136.

[107] Johan Muskens, R.J. Bril, and Michael R.V. Chaudron. Generalizing consistency checking between software views. *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) 2005*, 0, 2005. doi: http://doi.ieeecomputersociety.org/10.1109/WICSA.2005.37.

[108] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999. ISBN 3-540-65410-0.

[109] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) '98*, London, UK, 1998. Springer-Verlag. ISBN 3-540-64737-6.

[110] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17 (4), 1992. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/141874.141884.

[111] Andre Postma. A method for module architecture verification and its application on a large component-based system. *Information and Software Technology*, 45(4), 2003. URL `http://www.sciencedirect.com/science/article/B6V0B-47S70BV-1/2/7537942548b8aba316d369e8df646752`.

[112] Alex Potanin, James Noble, and Robert Biddle. Checking ownership and confinement. *Concurrency and Computation: Practice and Experience*, 16(7):671–687, April 2004. ISSN 1532-0634. doi: 10.1002/cpe.799. URL `http://dx.doi.org/10.1002/cpe.799`.

[113] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, and Larry Koved. Saber: smart analysis based error reduction. *SIGSOFT Software Engineering Notes*, 29(4), 2004. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1013886.1007545.

[114] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Julian Dolby, Aaron Kershenbaum, and Larry Koved. Validating structural properties of nested objects. In *Companion to the Proceedings of OOPSLA 2004*. ACM Press, 2004.

[115] Steven P. Reiss. An approach to incremental compilation. *SIGPLAN Not.*, 19(6):144–156, 1984. ISSN 0362-1340.

[116] Martin P. Robillard and Gail C. Murphy. Representing concerns in source code. *ACM TOSEM'07*, 16(1), 2007. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/1189748.1189751.

[117] Konstantinos Sagonas and Terrance Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM TOPLAS'98*, 20(3), 1998. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/291889.291897.

[118] Konstantinos Sagonas, Terrance Swift, and David S. Warren. Xsb as an efficient deductive database engine. *SIGMOD Rec.*, 23(2):442–453, 1994. ISSN 0163-5808. doi: http://doi.acm.org/10.1145/191843.191927.

[119] Diptikalyan Saha. *Incremental Evaluation of Tabled Logic Programs*. PhD thesis, Stony Brook University, 2006. URL `http://www.lmc.cs.sunysb.edu/~dsaha/diptithesis.ps`.

[120] Diptikalyan Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of PPDP '05*, New York, NY, USA, 2005. ACM Press. ISBN 1595930906. doi: 10.1145/1069774.1069785. URL `http://dx.doi.org/10.1145/1069774.1069785`.

[121] Jason Sawin and Atanas Rountev. Estimating the run-time progress of a call graph construction algorithm. In *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 53–62, 2006.

[122] Mohlalefi Sefika, Aamod Arvind Sane, and Roy Harrold Campbell. Monitoring compliance of a software system with its high-level design models. *Proceedings of the international conference on Software engineering (ICSE) 1996*, 1996. ISSN 0270-5257. doi: http://doi.ieeecomputersociety.org/10.1109/ICSE.1996.493433.

[123] Mariana Sharp, Jason Sawin, and Atanas Rountev. Building a whole-program type analysis in Eclipse. In *Eclipse Technology Exchange Workshop at the ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 6–10, 2005.

[124] Olin Shivers. Control flow analysis in scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 164–174, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: http://doi.acm.org/10.1145/53990.54007.

[125] Mati Shomrat and Amiram Yehudai. Obvious or not?: regulating architectural decisions using aspect-oriented programming. In *Proceedings of the international conference on aspect-oriented software development (AOSD) 2002*. ACM Press, 2002.

[126] Amie L. Souter and Lori L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pages 682–691, 2001. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=972787`.

[127] Richard M. Stallman and Roland McGrath. *GNU Make: A Program for Directing Recompilation*. Free Software Foundation, version 3.63 edition, January 1993.

[128] Martin Staudt and Matthias Jarke. Incremental maintenance of externally materialized views. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 75–86, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. ISBN 1-55860-382-4.

[129] Leon Sterling and Ehud Shapiro. *The art of Prolog*. MIT Press Cambridge, Mass, 1994.

[130] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. *ACM SIG-PLAN Notices*, 35(10):281–293, October 2000. ISSN 0362-1340. URL `http://www.acm.org/pubs/citations/proceedings/oops/353171/p281-tip/`. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[131] Tom Tourwe, Johan Brichau, Andy Kellens, and Kris Gybels. Induced intentional software views. *Computer Languages, Systems & Structures*, 30(1-2), 2004. URL `http://www.sciencedirect.com/science/article/B73H1-4BMC93H-1/2/45858c85dc089989147e6b0cd5243300`.

[132] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988.

[133] Jilles van Gurp and Jan Bosch. Design erosion: Problems and causes. *Journal of Systems and Software*, 61(2), 2002.

[134] Rob C. van Ommering, René L. Krikhaar, and Loe M. G. Feijs. Languages for formalizing, visualizing and verifying software architectures. *Comput. Lang.*, 27(1/3), 2001.

[135] Jan Vitek and Boris Bokowski. Confined types in java. *Software Pract. Exper.*, 31(6):507–532, May 2001. ISSN 0038-0644. doi: 10.1002/spe.369. URL `http://dx.doi.org/10.1002/spe.369`.

[136] Raphael Volz, Steffen Staab, and Boris Motik. Incremental maintenance of dynamic datalog programs (extended abstract). *Proc. of first Int. Workshop on Practical and Scalable Semantic Systems*, 2003. URL `citeseer.ifi.unizh.ch/654629.html`.

[137] Alan R. Williamson. *Ant: developer's handbook*. Sams, Indianapolis, IN, 2003. ISBN 0672324261.

[138] Tian Zhao, Jens Palsberg, and Jan Vitek. Lightweight confinement for featherweight java. *SIGPLAN Not.*, 38(11), 2003. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/949343.949318.